Multi-Paradigm Learning of Declarative Models

Cèsar Ferri Ramírez
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Memòria presentada per a optar al títol de:

Doctor en Informàtica

Directors:

José Hernàndez Orallo

María José Ramírez Quintana

València, 2003

Multi-Paradigm Learning of Declarative Models

Cèsar Ferri Ramírez
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

A thesis submitted for the degree of:

Doctor of Philosophy

Advisors:

José Hernàndez Orallo

María José Ramírez Quintana

València, 2003

Acknowledgements

La consecució d'aquesta tesi doctoral ha estat fruit de més de quatre anys d'intens treball d'investigació al Departament de Sistemes Informàtics i Computació de la Universitat Politècnica de València. És necessari ressaltar la desinteressada ajuda de moltes persones. Sense aquesta col·laboració la realització d'aquest treball haguera estat pràcticament impossible.

Primerament, voldria recordar Elvira Albert qui, gràcies a la seua amistat, va facilitar que s'iniciara aquesta aventura.

María Alpuente va ser qui va confiar en mi i em va oferir l'oportunitat de treballar al món de la investigació, dins el grup "Extensiones de la Programación Lógica", grup que lidera i dirigeix amb magnífics resultats.

No puc oblidar la generositat de Germán Vidal, qui es va esforçar en la meua formació com investigador, i a més, va fer que la meua integració al món universitari fóra molt més fàcil i agradable.

A continuació, voldria agrair la paciència i col·laboració de Santiago Escobar, qui ha estat company i amic durant aquests anys de treball. També vull recordar la gran qualitat humana i científica que he trobat dins el grup ELP: Salvador Lucas, Javier Oliver, Vicent Estruch, Alicia Villanueva, Francisco Correa, Ginés Moreno, Pasqual Julián i Ricardo Blanco. Companys que mai no han dubtat en donar-me l'ajuda que els he requerit.

Part dels treballs d'investigació han estat realitzats en estades d'investigació en centres estrangers. En concret voldria agrair a Michael Hanus i Frank Steiner la seua amabilitat per facilitar-me l'estada d'investigació a la Universitat de Kiel. Gràcies a la cordialitat i qualitat científica de Peter Flach vaig realitzar una estada molt fructífera a la Universitat de Bristol. Finalment, Moreno Falaschi i Marco Comini van possibilitar que la meua estada a la Universitat de Udine resultara un período que recorde molt gratament.

Aquestes estades a universitats estrangeres han estat realitzades gràcies a entitats com la Universitat Politècnica de València, la Generalitat Valenciana, el Gobern Espanyol i la Unió Europea, que m'han concedit suport econòmic mitjançant beques.

També voldria destacar l'ajuda que m'han prestat els companys del departament, tant professors, com tècnics o personal d'administració. En especial estic agrait a Antonio Molina, Antonio Vidal, Patricio Letelier, Carlos Monserrat i Mari Carmen Juan.

Una menció especial mereixen els meus directors de tesi:

- A María José Ramírez li he d'agrair tot l'interés i l'esforç que ha dedicat perquè aquesta tesi es duguera a terme. Gràcies al seu recolzament i al seu suport la meua tasca investigadora ha estat molt més senzilla.

- A José Hernàndez li vull reconéixer tot el temps i coneixement que m'ha oferit amablement durant aquests anys. A banda de la seua tasca investigadora, voldria ressaltar la seua gran paciència i altruisme, cosa que fa que treballar amb ell siga una experiència molt enriquidora. Finalment, a més d'un company exemplar, el considere un bon amic.

Tota la tesi s'ha realitzat amb software de lliure distribucuió amb llicencia GNU: GNU/Linux, LaTeX, cc, Weka, gnumeric, xemacs... Milers d'informàtics col·laboren desinteressadament en l'elaboració d'aquests programes. El software de lliure distribució facilita enormement el desenvolupament dels treballs per part de cientifics de tot el món.

Els meus amics a la Universitat: Pau Mendoza, Juan Pardo, Elena Rodilla i Nohemí Albuixech han contribuït que aquests anys de treball a la universitat es feren molts més amens i divertits.

Els meus pares i germans, qui em van animar a iniciar aquesta aventura i des d'aquell moment no han deixat de recolzar-me durant tot aquest trajecte.

Finalment, els meus esforços, dificultats i alegries han estat compartits per la meua nòvia Carme, qui m'ha oferit tota la seua paciència, comprensió i afecció que han permés que aquest camí es fera molt menys difícil amb la seua companyia.

Cèsar Ferri
València, 2003

Abstract

Many approaches for obtaining systems with intelligent behaviour are based on components that learn automatically from previous experience. The development of these learning techniques is the objective of the area of research known as machine learning. During the last decade, researchers have produced numerous and outstanding advances in this area, boosted by the successful application of machine learning techniques to knowledge discovery from databases, whose principal stage is referred to as data mining. Many techniques used in this field have very limited applicability due to the use of very restrictive or incomprehensible representation languages which are unable to express rich and understandable knowledge. This fact, however, significantly simplifies the inherent complexity of the learning process. For this reason, several methods which are based on the enrichment of language expressiveness have been recently defined. Specifically, the incorporation of first-order logic into the learning process has constituted a very important advance; this is indeed the nexus of the induction algorithms pertaining to the field called Inductive Logic Programming (ILP). However, first-order languages employed by ILP techniques lack some interesting features which are useful for learning: types, functions and higher-order features. Another approach to increasing the expressiveness of the representation language is the generation of a committee hypothesis by combining a set of models. These approaches, usually referred to as ensemble methods, however, have important disadvantages such as the loss of understandability of the combined model and a high consumption of computational resources.

In this thesis, we present some contributions to the field of machine learning that partially solve some of these problems. Specifically, we introduce learning methods that increase the expressiveness of the representation language, improve the accuracy of the learned models, and reduce the inherent costs of the learning process and the application of the learned model.

The first contribution is the definition of an induction framework based on multi-paradigm languages. These languages combine the best features of the two main families of declarative paradigms: functional programming and logic programming. The use of languages of this kind for learning provides advantages such as types and

functions. Our new learning framework has been implemented in a machine learning system, for which we include several experiments and results.

Learning with very expressive representation languages makes it possible to deal with problems having a complex structure. However, this considerably increases the search space of the learning algorithms, thus reducing their efficiency. Therefore, problems with a huge amount of data or attributes, typical of data mining, are better addressed by approaches based on simple representation languages such as propositional languages. The design of methods that could obtain comprehensible, and more accurate models according to the computational resources provided to the algorithm would be useful. In this context, this thesis introduces the definition of decision multi-trees and presents several applications for this structure. First, we define an *anytime* algorithm which utilises the multi-tree structure for the generation of decision trees, which gradually improve in terms of accuracy and comprehensibility with respect to classifiers constructed by the classic decision-tree learning algorithms. The multi-tree structure is also employed as an ensemble method by combining the hypotheses contained in the multi-tree. The major advantage of this method over other ensemble algorithms is the important saving in computational resources due to the very features of the multi-tree structure, where the hypotheses share common parts. In addition, we include a novel method for generating a comprehensible hypothesis that is semantically similar to the combined model, which requires even fewer resources.

The real usefulness of a learning method not only depends on the obtainment of expressive, comprehensible and accurate models, but also on the reduction of the costs related to the context in which they are going to be applied. Examples of these costs are misclassification costs or the costs necessary to apply the attribute tests in order to compute the predictions of a classifier (test costs). In this thesis, we introduce a technique based on the Area Under the *Receiver Operating Characteristic* (ROC) Curve (AUC), in order to efficiently evaluate the quality of a rule-based binary classifier independently of the cost context. We also use the efficiency of this measurement to derive a new splitting criterion that constructs decision trees based on ROC analysis. We extend this criterion to more than two classes, using approximations of the AUC for multi-class problems. Finally, we investigate several techniques to reduce the costs associated to the tests of classifiers, as well as study the integration of all the above techniques.

In summary, this thesis contributes new learning methods to generate more expressive, comprehensible and accurate models, which are capable of customising the use of computational resources, the quality of the classifiers and the costs of application in variable contexts.

Resum Estès

Moltes de les aproximacions per a aconseguir un comportament intel·ligent dels sistemes informàtics, es basen en components que aprenen automàticament d'experiències prèvies. El desenvolupament d'aquestes tècniques d'aprenentatge és l'objectiu de l'àrea d'investigació denominada com aprenentatge automàtic. Durant l'última dècada, s'han produït nombrosos i destacats avanços en aquesta àrea, principalment gràcies a l'èxit de l'aplicació de l'aprenentatge automàtic en l'extracció de coneixement des de bases de dades, la part més important de la qual s'anomena mineria de dades. Moltes de les tècniques emprades en aquest camp tenen una aplicabilitat molt limitada, per culpa de l'ús de llenguatges de representació de coneixement molt restrictius o no comprensibles, però què en canvi, simplifiquen significativament la complexitat inherent al procés d'aprenentatge. Per aquest motiu, recentment, s'han definit varios mètodes basats en enrequir l'expressivitat dels llenguatges de representació del coneixement. Concretament, un avanç molt important s'ha produït al incorporar llenguatges basats en la lògica de primer ordre en el procés d'aprenentatge; aquest és precisament el nexe comú dels algoritmes d'inducció que pertanyen al camp de la Programació Lògica Inductiva (ILP). Cal resaltar, no obstant, les carències que presenten els llenguatges lògics utilitzats en ILP en quant a característiques interessants per a l'aprenentatge, com ara l'ordre superior, els tipus o les funcions. Altra forma molt diferent, encara que també molt popular, de potenciar l'expressivitat del llenguatge de representació és la generació d'una hipòtesi mitjançant la combinació d'un conjunt de models. Aquesta aproximació, usualment anomenada multiclassificador, presenta importants inconvenients com són la pèrdua de comprensibilitat del model combinat i un elevat consum de recursos computacionals.

En aquesta tesi presentem varies aportacions al camp de l'aprenentatge que solventen parcialment alguns d'aquestos problemes. En concret, introduïm mètodes d'aprenentatge que augmenten l'expressivitat del llenguatge de representació, incrementen la precisió dels models apresos, i permeten la reducció dels costos inherents al procés d'aprenentatge.

La primera aportació és la definició d'un marc d'inducció basat en els llenguatges multiparadigma. Aquestos llenguatges aglutinen les millors característiques de les

dues famílies principals de paradigmes declaratius: els llenguatges funcionals i els llenguatges lògics. L'ús d'aquestos tipus de llenguatges en l'aprenentatge presenta una sèrie d'advantatges com ara, la utilització directa de tipus i de funcions. Aquest marc d'inducció ha estat implementat en un sistema d'aprenentatge automàtic, del qual incloem diversos experiments i resultats.

L'aprenentatge amb llenguatges de representació molt expressius permet el tractament de problemes amb una estructura complexa; en canvi, augmenten considerablement l'espai de cerca dels algoritmes d'aprenentage, reduint la seua eficiència. Per tant, els problemes amb moltes dades i atributs, típics en la mineria de dades, s'aborden mitjançant algoritmes basats en llenguatges de representació més simples, com ara els llenguatges proposicionals. Sembla interessant dissenyar mètodes que permeten obtenir models comprensibles, i que siguen més precisos quant més recursos computacionals es proporcionen a l'algoritme. En aquest context, aquesta tesi aporta la definició de multiarbres de decisió sobre els quals presentem diferents aplicacions. Primer definim un algoritme *anytime* que empra aquesta estructura multiarbre per a la generació d'arbres de decisió. L'estructura multiarbre l'usem també com a mètode multiclassificador, és a dir, combinant les hipótesis presents en el multiarbre. El principal advantatge d'aquest mètode respecte d'altres algoritmes multiclassificadors és un important estalvi dels recursos computacionals, gràcies a les característiques del multiarbre, on les hipòtesis comparteixen part de la seua estructura. Tant mateix, presentem un mètode novedós per a generar una hipótesi que s'aproxima semànticament al model combinat, però amb l'advantatge de què aquesta hipòtesi és comprensible, i a més a més, per al seu ús i emmagatzemament es requereix menys recursos.

La vertadera utilitat d'un mètode d'aprenentatge no sols depén de l'obtenció de models expressius, comprensibles i precisos, sinó que també és vital per a reduir els costos relacionats amb el context on es va a utilitzar. Exemples d'aquestos costos, són els costos de l'error en una predicció, o bé, els costos necessaris per a realitzar proves (tests) que permeten al model efectuar prediccions. En aquesta tesi introduïm una tècnica basada en l'àrea (AUC) sota la corba *Receiver Operating Characteristic* (ROC), la qual permet avaluar eficientment la qualitat d'un classificador binari independentment del context de costos. L'eficiència d'aquesta mesura ens serveix per a la definició d'un nou criteri de partició que permet la construcció d'arbres de decisió basats en l'anàlisi ROC. Extenem aquest criteri a més de dues classes, utilitzant aproximacions de l'AUC per a problemes multiclasse. Per últim, també investiguem diverses tècniques que serveixen per a reduir els costos associats a la realització dels tests.

En resum, aquesta tesi aporta noves tècniques d'aprenentatge per a generar models més expressius, comprensibles i precisos mitjançant un menor ús dels recursos computacionals en contextos variables.

Resumen Extendido

Muchas de las aproximaciones para conseguir un comportamiento inteligente de los sistemas informáticos se basan en componentes que aprenden automáticamente de experiencias previas. El desarrollo de estas técnicas de aprendizaje es el objetivo del área de investigación denominada aprendizaje automático. Durante la última década, se han producido numerosos y destacados avances en esta área, principalmente gracias al éxito de la aplicación del aprendizaje automático en la extracción de conocimiento desde bases de datos, a cuya parte más importante se le llama minería de datos. Muchas de las técnicas empleadas en este campo tienen una aplicabilidad muy limitada, debido al uso de lenguajes de representación del conocimiento muy restrictivos o no comprensibles, pero que, sin embargo, simplifican significativamente la complejidad inherente al proceso de aprendizaje. Por ello recientemente, se han definido varios métodos basados en el enriquecimiento de la expresividad de los lenguajes de representación del conocimiento. En concreto, un avance muy importante se ha producido al incorporar lenguajes basados en la lógica de primer orden en el proceso de aprendizaje; éste es precisamente el nexo común de los algoritmos de inducción pertenecientes al campo denominado Programación Lógica Inductiva (ILP). Cabe resaltar, no obstante, las carencias que presentan los lenguajes lógicos utilizados en ILP en cuanto a características interesantes para el aprendizaje, tales como el orden superior, los tipos, o las funciones. Otra forma muy diferente, aunque también muy popular, de potenciar la expresividad del lenguaje de representación es la generación de una hipótesis mediante la combinación de un conjunto de modelos. Esta aproximación, usualmente denominada métodos multiclasificador, sin embargo presenta importantes inconvenientes como son la pérdida de comprensibilidad del modelo combinado y un alto consumo de recursos computacionales.

En esta tesis presentamos varias aportaciones al campo del aprendizaje que solventan parcialmente algunos de estos problemas. En concreto, introducimos métodos de aprendizaje que aumentan la expresividad del lenguaje de representación, incrementan la precisión de los modelos aprendidos, y permiten la reducción de los costes inherentes al proceso de aprendizaje.

La primera aportación es la definición de un marco de inducción basado en los lenguajes multiparadigma. Estos lenguajes aglutinan las mejores características de las dos familias principales de paradigmas declarativos: los lenguajes funcionales y los lenguajes lógicos. El empleo de este tipo de lenguajes en el aprendizaje presenta una serie de ventajas tales como la utilización directa de tipos y funciones. Este marco de inducción ha sido implementado en un sistema de aprendizaje automático, del cual incluimos diversos experimentos y resultados.

El aprendizaje con lenguajes de representación muy expresivos permite el tratamiento de problemas con estructura compleja; sin embargo, aumenta considerablemente el espacio de búsqueda de los algoritmos de aprendizaje, reduciéndose su eficiencia. Por lo tanto, los problemas con muchos datos y atributos, típicos en la minería de datos, se abordan con algoritmos basados en lenguajes de representación más simples, como los lenguajes proposicionales. Parece interesante diseñar métodos que permitan obtener modelos comprensibles, y que sean más precisos cuanto más recursos computacionales se proporcionen al algoritmo. En este contexto, esta tesis aporta la definición de multiárboles de decisión sobre los cuales presentamos diferentes aplicaciones. Primero definimos un algoritmo *anytime* que emplea esta estructura multiárbol para la generación de árboles de decisión que mejoran paulatinamente en precisión y comprensibilidad los algoritmos clásicos de aprendizaje de árboles de decisión. La estructura multiárbol la usamos también como método multiclasificador, es decir, combinando las hipótesis presentes en el multiárbol. La principal ventaja de este método respecto a otros algoritmos multiclasificadores es un importante ahorro en recursos computacionales debido a las características del multiárbol, donde las hipótesis comparten parte de su estructura. Asimismo, presentamos un método novedoso para generar una hipótesis que se parece semánticamente al modelo combinado, pero con la ventaja de que esta hipótesis es comprensible, y, además, para su uso y almacenamiento se requieren todavía menos recursos.

La verdadera utilidad de un método de aprendizaje no sólo depende de la obtención de modelos expresivos, comprensibles y precisos, sino que es también vital reducir los costes relacionados con el contexto en el que se van a aplicar. Ejemplos de estos costes son los costes del error en una predicción, o bien los costes necesarios para realizar pruebas (tests) que permitan al modelo efectuar predicciones. En esta tesis introducimos una técnica basada en el área (AUC) bajo la curva *Receiver Operating Characteristic* (ROC), que permite evaluar eficientemente la calidad de un clasificador binario independientemente del contexto de costes. La eficiencia de esta medida nos sirve para la definición de un nuevo criterio de partición que permite la construcción de árboles de decisión basado en el análisis ROC. Extendemos dicho criterio a más de dos clases, utilizando aproximaciones del AUC para problemas multiclase. Por último, también investigamos diversas técnicas que sirven para reducir los costes asociados a la realización de los tests.

En resumen, esta tesis aporta nuevas técnicas de aprendizaje para generar modelos más expresivos, comprensibles, precisos y con un uso menor de recursos computacionales en contextos variables.

A Enric i Pau.

A Carme.

The real problem is not whether machines think but whether men do.

Burrhus F. Skinner

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Nowadays, most companies, governments and institutions deal with huge amounts of data. This information is very useful for predicting future events or for explaining past facts. In the best case, accurate models of a parcel of reality can be constructed and then they can be used to make decisions according to the model's predictions. Because of the large amount of information available and because of the complexity of many concepts that need to be handled, computational models have been proved in many cases to make better and more effective decisions than humans.

During the last few years, there have been significant advances in the extraction of knowledge from data, mainly due to the use of machine learning techniques. This has converted the classical area of data analysis into the new areas of knowledge discovery and data mining. Machine learning researchers have been working on the automatic generation of general models from a set of data, trying to automate inductive learning, i.e., the reasoning process from the specific to the general.

Machine learning has been defined by Mitchell [Mitchell, 1997] as the area of computer science that is concerned with the question of how to construct computer programs that automatically improve with experience. Recently, there have been important advances in the theoretical foundations of this field. At the same time, many successful applications have been developed: systems for extracting information from databases (data mining), applications to support decisions in medicine, telephone fraud and network intrusion detection, prediction of natural disasters, game playing, email filtering, document classification, and many others.

Although inductive learning is generally devoted to identifying patterns and regularities from data, there are several different kinds of learning, depending on the presentation of the problem. Supervised learning is the subarea of machine learning that studies and develops learning systems that produce predictions for different inputs based on input-output pairs (labelled examples) that have been previously perceived. When the set of possible predictions is finite, the task is called *classification*. When the set of possible predictions is infinitely ordered, the task is called *regression*. Unsupervised learning works with unlabelled examples aimed at describing the data. *Clustering*, *association rules* and *correlations* are typical unsupervised tasks.

The range of techniques applied to inductive learning is very extensive. However, many of the techniques behave like black-box systems, i.e., models that produce an output from an input, but whose behaviour is hard to understand, such as artificial neural networks. In many situations, the person or organisation that is making a decision needs to understand the model produced by the computers. Therefore, it is not only important to produce good models, but their comprehensibility is also essential.

From the set of different techniques developed by the machine learning discipline, comprehensibility is the major advantage that has been advocated for supporting certain machine learning methods such as decision-tree learning, rule learning or inductive logic programming. One major feature of what is called "Knowledge Discovery" is that it must give insight from the models, properties and theories that are obtained. A model that is not comprehensible may be useful for obtaining good predictions, but it cannot provide knowledge about how predictions are made.

Apart from comprehensibility, another important factor in the learning process is the expressive power of the representation language employed in the evidence and the models. Classical machine learning methods (propositional learning) are very restrictive due to the use of limited representation languages. Methods of this kind utilise very simple and restrictive rules (propositional rules) as the representation language, and therefore, learning from complex and structured data becomes difficult. Relational learning, specifically *Inductive Logic Programming* (ILP), represents an important advance in expressive power. ILP uses logic programming as the representation language. However, logic programming lacks some interesting features (such as lazy evaluation strategies, types, higher order features and functions) which are not supported by logic programming.

Recently, after almost two decades of research, the declarative programming community has defined and implemented another powerful and useful programming paradigm: functional logic programming. This paradigm combines the most important features of functional programming and logic programming. Therefore, functional logic programming represents an important step forward as a programming paradigm that supports the best features of declarative languages. However, due to the relative newness of the functional logic paradigm, it has not been widely used for learning despite the very important advantages that this paradigm offers.

Another method which is employed to increase accuracy is the combination of a set of models, or ensemble. The process of combining an ensemble of models can also be interpreted as a strategy to increase the expressive power of the representation language. Multi-classifier systems usually improve accuracy significantly. Nevertheless, they also introduce important disadvantages. They increase the required resources during learning and also during the application of models and more importantly, the final hypothesis is a combination of a set of models. Thus, the comprehensibility of the final model is completely lost.

Although comprehensibility, accuracy and expressiveness may be the most important factors for the usability of machine learning techniques, there are other drawbacks or costs associated with the learning process. For instance, in many situations not every error has the same consequences. In classification tasks, for instance, some errors or some conditions may have greater costs than others, especially in diagnosis. Cost-sensitive learning is hence a generalisation of learning, allowing models to make better decisions.

When misclassification costs are unknown at learning time, ROC analysis offers the possibility of selecting the subset of models that will be optimal in any misclassification-cost context.

Therefore, it is important for machine learning techniques to be able to produce models that not only minimise the prediction error, but that also distribute the errors appropriately according to their misclassification costs.

In summary, ILP methods are usually inefficient and do not take costs into consideration, especially in multi-class problems (partly because of the lack of functions). They are less accurate than other approaches, such as ensemble methods. The application of ensemble methods to decision-tree learning has led to the loss of comprehensibility of the induced models.

As a result of all the above considerations, it then becomes necessary to develop new methods and paradigms for the learning of comprehensible models that not only minimise the number of mistakes, but also minimise the costs associated with those decisions.

The definition of machine learning methods which are able to deal with all these features (expressiveness, comprehensibility and cost-sensitivity) has been very limited until recently. This is mainly because the conjunction of all these features introduces a high degree of complexity into the learning process, thereby increasing the computational cost significantly.

1.2 Objective of this Dissertation

The general purpose of this work is the design and development of new learning methods and paradigms for obtaining accurate and comprehensible models in the form of equational rules from a set of past experiences. More specifically, the methods to be developed generate models that behave well in contexts where there are misclassification costs, or where the class distribution of the examples are uncertain. The constructed models are represented by declarative languages for the understandability of the users.

For this purpose, we undertake three main lines of research:

1. More Expressive Models

   ILP has represented an important step forward in machine learning, mainly due to the introduction of very expressive languages, namely first-order logic, in the learning process. However, languages of this kind lack some attractive features such as functions or higher-order facilities.

   These limitations have been overcome by modern multi-paradigm languages such as functional logic programming. Nevertheless, languages of this kind have not been widely used for learning purposes to date.

   The first goal of this dissertation is the definition of a framework for learning functional logic programs and the implementation of a prototype system based on this definition. In this approach, evidence, programs and background knowledge will be expressed over a functional logic language. The framework will represent a natural extension of the inductive logic programming paradigm to functional logic languages. We will explore the potential advantages of the use of functional logic programs, paying special attention to the rich expressiveness that functional features provide.

2. More Accurate Comprehensible Models

   In contexts where there are huge amounts of data, or the structure is not complex, relational learning methods are not very appropriate due to their low efficiency. The main cause is the wide search space that is introduced by the use of rich representation languages.

   A different way of capturing complex models is the use of ensemble methods. In some situations, it is more efficient and accurate to use ensembles than to use ILP systems. However, the main drawback of multi-classifiers methods is that the comprehensibility of the learned model is lost.

   The second goal of this dissertation is to introduce new learning methods that can use the power of ensemble methods to generate models that are more accurate than classical decision-tree learners, but are still representable in the form of comprehensible rules. The results of this process are equational rules which are represented by decision trees.

3. Less Expensive Models

   Comprehensibility, expressiveness and accuracy are not always enough for a model to be useful. The learning process involves different and important types of costs, such as misclassification costs or test costs.

   Our third goal is the development of learning techniques based on the above approaches that are able to behave well in any misclassification-cost context. We also study some methods to reduce the test cost of models. Consequently, learning is re-designed in order to minimise cost rather than to minimise error, and this affects the way in which the results of the methods presented here are evaluated.

1.3 Organisation of this Dissertation

This dissertation is organised in four parts:

- Part I: State of the Art

  Part I presents the state of the art on the topics of machine learning in which this work is focussed.

  First, in Chapter 2 we review the basics of propositional learning: definition, features and major methods. Since a part of the dissertation is based on decision-tree learning, we pay special attention to describing this method. We conclude the chapter by presenting the important limitations in expressiveness of propositional learning.

  Chapter 3 is devoted to relational learning, specifically Inductive Logic Programming. We introduce the main characteristics and applications of this paradigm, as well as the most important methods. We also discuss the deficiencies of logic programming as a representation language for learning.

  The foundations of functional logic programming are included in Chapter 4. This chapter contains the theoretical basis and a general description of this paradigm. We also examine some issues about the use of functional logic programs as a representation language for expressing learning models.

  In Chapter 5, we include an introduction to cost-sensitive learning. Specifically, we centre on misclassification costs and on test costs. We concentrate on ROC analysis and evaluation measures derived from it, such as the Area Under the ROC Curve (AUC).

Finally, Part I concludes with Chapter 6. In this chapter, we summarise the most relevant features of ensemble methods. We describe the most important techniques for constructing and combining ensembles, as well as the use of ensemble methods in relational learning. Finally, we present a discussion of the problems and limitations of these techniques.

- Part II: Induction of Functional Logic Programming

  Part II defines a new framework for the automatic induction of functional logic programs (IFLP). First, Chapter 7 introduces the operators on which the IFLP approach is based. We define an algorithm to induce programs automatically that employs the IFLP operators. We also present an incremental extension of this method. This extension makes it possible to increase the efficiency of the technique. Then, Chapter 8 describes the FLIP system, a prototype system that implements the IFLP framework. This system can automatically induce functional logic programs from evidence expressed as ground equations. It can also deal with background knowledge (in this case functional logic programs). Some experimental results show the usefulness of the system. Finally, we discuss some interesting applications.

- Part III: Decision Multi-trees

  Part III introduces some innovations in ensemble learning: the sharing of parts between hypotheses, an anytime construction policy, and several single decision-tree extraction methods. In Chapter 9, we present a method based on decision trees that permits the anytime generation of a set of models which share part of their structure. For this purpose, we employ a beam search over a structure called decision multi-tree. Chapter 10 addresses how to combine the elements of the multi-tree. This constitutes an ensemble method, which has important advantages over classical ensemble methods. Lastly, we present a technique that allows us to derive an accurate and comprehensible model from the ensemble of models, thereby bringing comprehensibility to ensemble methods.

- Part IV: Cost-sensitive Learning

  Finally, we address the issue of cost-sensitive learning. Chapter 11 includes some results on multi-class ROC analysis; the development of derived measures that can be computed efficiently and their application to the cost-sensitive learning of decision trees. We introduce some new splitting criteria, which are based on well known measures that allow us to construct decision trees that behave well in any misclassification-cost context. The minimisation of test costs is also addressed in this chapter. We illustrate some techniques that use the multi-tree structure and that allow us to minimise the test cost of the generated models.

The dissertation concludes with Chapter 12, where we summarise the main contributions of the work and where we outline several possible directions for further research.

Finally, we add some appendices with additional information. We include a complete description of the datasets employed in the dissertation. We also incorporate a detailed description of the systems on which the theoretical concepts of this dissertation have been implemented: FLIP and SMILES.

Part I

State of the Art

Chapter 2

Propositional Learning

In this chapter, we include a review of the most important methods of machine learning that employ propositional languages for representation of knowledge. Despite the inherent limitations of these methods due to the restrictive expressiveness of propositional rules, nowadays they are widely employed because their high efficiency allows them to address problems with a great number of instances. Since part of the thesis employs decision-tree learning, we pay special attention to this learning technique, describing the most important strategies to construct decision trees. Finally, we discuss the important limitations of using proportional rules for dealing with problems with complex structure.

2.1 Introduction

Machine learning provides techniques that allow models to be generated from past experiences. During the last few years, several learning techniques have been developed and used for many practical applications. Some of these techniques are decision trees [Breiman *et al.*, 1984], decision rules [Michalski, 1983], neural networks [Rumelhart *et al.*, 1986], Bayesian classification and statistical methods [Buntine, 1990], genetic algorithms [Booker *et al.*, 1989], inductive logic programming [Muggleton, 1991; Lavrac and Dzeroski, 1994; Muggleton, 1999], ensemble learning [Dietterich, 2000a], and support vector machines [Cristianini and Schöllkopf, 2000].

In this dissertation we address aspects of supervised learning, the subarea of machine learning where the learning algorithm is provided with a set of inputs along with the corresponding correct outputs. Hence, learning requires to compare its current actual output with the correct or target outputs, and modify things accordingly [Wilson, 1998]. Formally, the input of supervised learning is a set of labelled examples (or *instances*) $\langle x_i, y_i \rangle$ known as *training* dataset, where x_i is a vector of continuous or discrete values called *attributes* and y_i is the *label* of x_i. The training dataset is assumed to be generated according to a probability distribution $P(x, y)$ which is usually unknown. If the labels are elements of a discrete set of classes, the supervised learning is called *classification*; if they are elements drawn from a continuous set, it is called *regression*.

Assuming that the examples in the training dataset are drawn from an unknown probability distribution, the goal of supervised learning is to produce a mapping h (called *hypothesis)* from attributes to labels, which will be able to label correctly a high proportion of unlabelled examples drawn from the same distribution. Therefore, the learning algorithm describes a procedure for building a model, such that the model will implement a hypothesis $h(x)$ that minimise the error labelling any example drawn according to the same probability distribution.

In many cases, it is very useful to learn models as a set of comprehensible rules. The rules can be understood, improved, and validated by humans, and then can be applied by to users to predict future events and to make decisions. The models can even be considered as an explanation of a parcel of reality. A model that is not comprehensible may be useful to obtain good predictions but it cannot provide knowledge about how predictions are made.

There are many methods in machine learning that allow a comprehensible set of rules to be generated from an evidence. These methods can be divided into two groups depending on the representation language they employ. The first group is formed by decision tree learning and classical rule learning methods; all these employ propositional rules. The propositional rules have a very restrictive expressiveness because they cannot contain variables. A second group is composed by methods that use very expressive representation languages like first-order Horn clauses. This language can represent very expressive rules with advanced features such as variables, predicates and function calls. This second family is usually called Inductive Logic Programming (ILP).

Table 2.1 presents a classification of the most important machine learning techniques depending on the comprehensibility of the models that they produce, and depending on their expressive power. We refer to [Mitchell, 1997] for details about the techniques shown in the table.

Since recent years, decision tree learning has been successfully employed for the Data Mining part of the Knowledge Discovery in Databases (KDD) [Fayyad, 2001]. KDD is defined as: *"the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data"* [Fayyad, 2001]. Since more and more information is being stored in databases, the sources and consequently applications of data mining are pervading many fields: from medicine to decision support systems (DSS). Data Mining is just a part of this complex process; KDD includes Data

	Propositional/Linear	Relational/Universal
Comprehensible	Decision Trees Rule Learning	Inductive Logic Programming (ILP)
Incomprehensible or without model	Nearest Neighbour (K-NN) Bayes Classifiers Perceptron Kmeans	Artificial Neural Networks (ANN) Support Vector Machines (SVM) Kernel methods

Table 2.1: Machine learning methods.

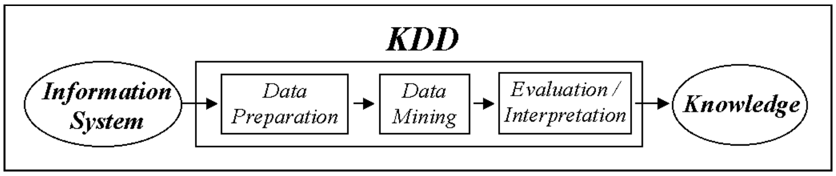


Figure 2.1: KDD Process.

Preparation, Data Mining itself and Interpretation/Evaluation, as shown in Figure 2.1.

The purpose of this chapter is to present the most important methods of propositional learning that produce comprehensible methods. Concretely, we introduce decision tree learning and rule learning. We also discuss the applications and limitations of these approaches.

2.2 Decision Trees

Decision tree learning is one of the most widely used techniques in machine learning. The objective is to approximate a target function, in which the learned function is represented by a decision tree. Learned trees can also be represented as sets of *if-then* rules to improve their readability. Because of this, decision tree learning is among the most popular of all the inductive inference algorithms and it has been successfully employed in many applications. Decision trees have been employed for classification [Quinlan, 1993], regression [Breiman *et al.*, 1984], and clustering [Liu *et al.*, 2000]. In this chapter, we will present decision tree learning for classification.

Formally, a decision tree is a tree where each node in the tree specifies a test on some attribute of the function, and each branch descending from that node corresponds to the outcome value for this attribute. Finally, every leaf node of the tree includes a class prediction.

An example is classified by starting at the root node of the tree, testing the attribute specified by the node, then following with the tree branch corresponding to the value of the attribute in the given example. This process is repeated until the node to be tested is a leaf node. Then, the leaf node contains a class prediction for this example.

Figure 2.2 shows a decision tree for the classical *playtennis* example [Mitchell, 1997]. The leaf nodes are surrounded by a double square. From this tree, 5 rules could be easily extracted. Each rule is by the conditions of the branches in the path from the root to each leaf node of the decision tree:

$$\begin{array}{lcl} playtennis(rain, X2, X3, strong) & = & no \\ playtennis(rain, X2, X3, weak) & = & yes \\ playtennis(overcast, X2, X3, weak) & = & yes \\ playtennis(sunny, X2, high, X4) & = & no \\ playtennis(sunny, X2, normal, X4) & = & yes \end{array}$$

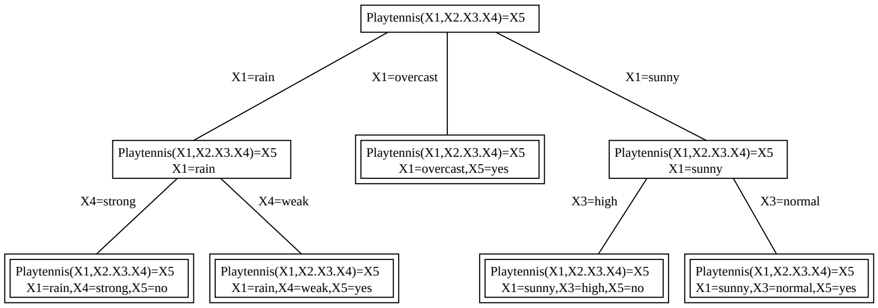


Figure 2.2: Decision tree for the *playtennis* example.

2.2.1 Decision Tree Learning

Decision tree learning was introduced by Breiman [Breiman *et al.*, 1984], and later on Quinlan developed some of the most well-known systems for learning decision trees: ID3 [Quinlan, 1990a], C4.5 [Quinlan, 1993], and C5.0 [Quinlan, 1998a].

Most algorithms that generate decision trees to classify instances employ a greedy top-down search. This is the case of CART [Breiman *et al.*, 1984], ID3 [Quinlan, 1990a], C4.5 [Quinlan, 1993] and DKM [Kearns and Mansour, 1996].

In these basic algorithms, the essential issue to build effective decision trees is the *splitting criterion*, i.e. which attribute should be tested in each step of the process. To select the attribute, each possibility is evaluated using a statistical test to determine the quality of the partial classification. The best attribute is selected and used as the test at the root node of the tree. A descendant of the root is then generated for each possible value. This process is repeated for each new descendant node. Therefore, the decision tree is constructed following a greedy search.

In general, splitting criteria find the split s with the lowest $I(s)$, where $I(s)$ is defined as:

$$I(s) = \sum_{j=1..n_j} p_j \cdot f(p_j^+, p_j^-)$$

where n_j is the number of alternatives in the split, p_j is the probability that an example falls into the j-th node (the cardinality of the child node divided by the cardinality of the parent node), p_j^+ is the proportion of negative examples in s, and p_j^- is the proportion of positive examples in s . Using this general formula, each splitting criterion implements a different function f, as is shown in the following table:

Criterion	f(a,b)
Accuracy(EError)	min(a,b)
Gini(CART)	$2ab$
Entropy(Gain, ID3)	$a \cdot \log(a) + b \cdot \log(b)$
DKM	$2(a \cdot b)^{1/2}$

Since these functions $f(a,b)$ are impurity functions, $I(s)$ calculates a weighted average of the impurity of the children in s.

Originally, the first decision tree learning systems were only able to deal with examples formed by discrete arguments. However, later extensions have widely amplified its applicability. C4.5 can learn decision trees with examples containing continuous-valued attributes (but the class must still be a nominal value). CART can also construct decision trees for a continuous-valued class (*regression*).

Other interesting extension is the possibility of learning decision trees from data with missing values for some attributes. Some methods that consider these situations in the decision tree learning process have been defined [Mingers, 1988; Quinlan, 1993]. For instance, C4.5 employs a method that assigns to the unknown values of an attribute new values based on the probabilities of the known examples of that attribute.

Pruning the decision tree

Decision tree learning, such as has been described, grows each branch of the tree until each node could be considered as a leaf, i.e. when the training examples that fall into that node have the same class. This is not the best strategy when the training examples contain noisy data, i.e. examples with mistakes, or when the size of the evidence is too small to be considered as a representative sample of the true target function. In these cases, the original decision tree learning algorithm *overfits* the training examples. It is said that a hypothesis overfits the training data if some other hypothesis that fits training examples worse actually performs better over the entire distribution of instances [Mitchell, 1997].

For avoiding this overfitting problem in decision tree learning some techniques have been introduced. Most of them try to *prune* the decision tree by removing some lower branches that are considered to be much too specific. This process modifies the structure of the tree by reducing its size.

Depending on when the pruning process is performed, we can divide the pruning techniques into to main groups:

- **Pre-pruning:** A pruning method is considered *pre-pruning* when it performs the analysis and the removal of the branches within the process of the construction of the decision tree. Hence, these methods must be included in the decision tree construction phase.

- **Post-pruning:** A *post-pruning* algorithm inspects and reduces a decision tree to avoid overfitting once the whole decision tree is completed. In this sense, it can be seen as a later process to the construction phase.

Pre-pruning is usually more efficient although post-pruning can be more accurate, since the whole picture of the tree is available. There are many pruning (post-pruning and pre-pruning) methods based on different measures. For instance, the *Minimum Description Length* principle MDL has been employed in some pruning approaches [Quinlan and Rivest, 1989; Mehta *et al.*, 1995]. C4.5 employs a post-pruning method based on the performance evaluation of the training set, using a pessimistic estimate [Quinlan, 1993].

Some empirical evaluations that have compared the efficiency and quality of pruning methods are [Mingers, 1989; Esposito *et al.*, 1997].

2.2.2 Applications

Decision trees learning is a suitable technique for problems where instances are represented by attribute-value pairs. It can also deal with errors in the data, and with missing attribute values. As we have stated, the induced tree can also be seen as a set of rules. Hence, it is a method which produces comprehensible models. Due to its greedy behaviour, it can deal with medium, even large, problems, i.e. problems with a great amount of examples, or with many attributes.

Many practical problems have been found appropriate for the use of decision tree as a learning method. Decision tree learning has therefore been applied to a wide range of problems: in medical environments to classify patients according to their disease, for banking environments to know the risk level of a loan applicant or insurance applicants by their life conditions, extracting information from databases, telephone fraud and network intrusion detection, and document classification.

2.3 Rule Learning

A major advantage of decision tree learning is the possibility of representing the decision tree as a set of equivalent *if-then-else* rules. These rules allow the user to understand the behaviour of the learned function.

There are other approaches that can directly construct a set of propositional rules from an evidence composed by a set of examples. These methods are called rule learners.

Most of the rule learners employ a sequential strategy to construct the set of rules. This strategy consists in learning one rule, removing the data it covers, then iterating this process. The procedure is repeated until the set of rules covers an acceptable portion of the evidence. These algorithms are called *sequential covering* algorithms. This process reduces the problem of learning a set of rules to a sequence of simpler problems, each requiring that a single rule be learned. These methods are based on the family of algorithms called AQ [Michalski, 1969; Michalski *et al.*, 1986].

Therefore, an important issue for this kind of algorithms is how to learn a simple rule. This process is usually performed in a similar way to classical decision tree learners. The search begins by considering the most general rule precondition possible: the empty test. Then, some conditions are added to the rule according to a rule performance measure over the training examples. This process of addition of conditions is performed until an acceptable level of performance is reached. The main difference of this process with respect to decision tree learning is that when a test is selected just one descendant of the test is followed rather than growing a subtree that covers all possible values of the selected attribute.

The sequential covering algorithm performs a greedy depth-first search of every rule. A drawback of a greedy search is that it can make a suboptimal chance at any step of the process. Some versions have been proposed to overcome this problem. For instance, CN2 [Clark and Niblett, 1989] performs a *beam search*, i.e., a search in which the algorithm maintains a list of k candidates at each selection, instead of considering just a single option. On each search step, the algorithm generates descendants from these k candidates, and the resulting set is again reduced to the k most promising members. Despite the use of the beam search to reduce the risk, the greedy search may still produce suboptimal rules.

2.4 Representation Limitations of Propositional Learning

Decision tree learning and rule learning constitute two very powerful methods for learning propositional rules. These methods, as we have stated, have been applied in many practical problems. However, many other problems exist for which the propositional expressiveness is insufficient.

Consider, for instance, the following problem from [Michie *et al.*, 1994; Flach, 2000]. The objective is to classify trains as Eastbound or Westbound. The evidence is shown in Figure 2.3.

Each train can contain a different number of cars. Each car has a different *shape* (rectangular, oval, u-shaped, ...), *length* (long, short), *number of wheels* (2, 3), *type of roof* (none, jagged, peaked, ...), *shape of load* (circle, triangle, rectangle, ...), and *number of loads* (1-3). A possible model to classify the examples is this: a train that contains a short closed car is eastbound, and westbound otherwise.

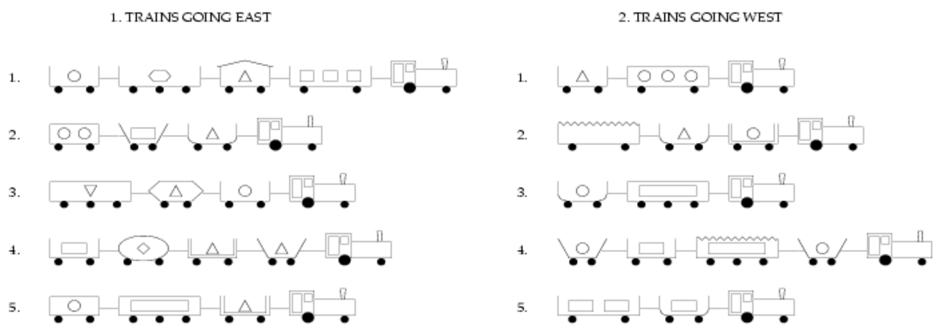


Figure 2.3: Evidence for the trains problem.

The representation of this problem in a propositional framework presents several and important drawbacks. [De Raedt, 1998; Flach, 2000] presents the main problems of representing complex scenes by a fixed set of attribute-value pairs:

1. One should reckon the maximum number of attributes that could appear for every scene. For some examples, some of the attributes will have a nil value, since not all the scenes contain the same number of objects. For the trains scene, we need 24 for all the examples because we represent with 6 attributes each car, and there are some trains with 4 cars. Note that the trains with less than 4 cars must complete the 24 attributes, therefore some of them will be nil values, and trains with more than 4 could not be handled.

2. One should order the objects in the examples. Without determining the order of the objects in a scene, there is an exponential number of equivalent representations of a problem (with respect to the number of objects).

There are many real-life application areas where these representational issues are frequent, e.g., the field of molecular biology [Bratko *et al.*, 1998] where the data are formed by set of molecules with a very complex structure and properties, or the analysis of proteins [Turcotte *et al.*, 1998].

Other important problem of propositional methods is the impossibility of the use of previously known information (*background knowledge*). This knowledge could be expressed with auxiliary functions that can be incorporated to the target function during the learning process. This limitation precludes the learning of complex problems where the application of auxiliary knowledge is essential. For instance, consider the learning of a function for the product of two natural numbers. This goal could be hardly reached without the help of a function for the addition of natural numbers.

Finally, another important problem is that, intrinsically, propositional models are unable to capture many concepts. For instance, the problem of determining if a number is even or not, or the monks1 problem. The later is a popular problem from the UCI ML dataset repository [Blake and Merz, 1998] which defines the function $monks1$, such that $monks1(\_,\_,\_,\_,1,\_)$ and $monks1(X,X,\_,\_,\_,\_)$ are $true$.

Chapter 3

First-order Learning

In this chapter, we offer a general overview of the state of the art in one of the areas of interest of this thesis: the use of first order languages for machine learning.

3.1 Introduction

The first generation of techniques applied in machine learning were mainly propositional, in the sense that they were only able to deal with flat, well structured data in one table. These methods can establish of conditions among different attributes of one example, but they cannot find relationships among different examples, as we have shown in the previous chapter. The use of recursion and background knowledge for learning was also restricted.

With the goal of overcoming these important restrictions, some works were developed during the last two decades that opened a new and innovative field in machine learning. The common base of these works was the use of very expressive and comprehensible models, using first-order logic as representation language. The pioneer of this field was Shapiro with his classical model inference system [Shapiro, 1983], where he introduced some innovative ideas in the field of Logic Programming, such as induction or debugging.

To show the usefulness of first-order languages for representing complex problems, let us refer to the trains problem presented in Section 2.4. In that section, we illustrated the limitations and drawbacks of using propositional languages, i.e., a fixed set of attribute value pairs. In a first-order language like Prolog, each example can be represented by a fact. The following representation of a train uses functors to represent cars, tuples for the loads, and lists to represent the sequence of cars. For instance, the following fact represents the first train in Figure 2.3.

$$\begin{aligned} eastbound \quad & ([car(rect, short, none, 2, load(circ, 1)), \\ & car(rect, long, none, 3, load(hexa, 1)), \\ & car(rect, short, peak, 2, load(tria, 1)), \\ & car(rect, long, none, 2, load(rect, 3))]). \end{aligned}$$

By using this representation the hypothesis can be expressed as follows:

$$eastbound(T) : -member(C, T), arg(2, C, short), not\ arg(3, C, none).$$

where predicate $member/2$ returns each member of a list, predicate $arg/3$ returns the component of a specified position in a term, and $not/0$ is the negation. Note that these predicates are assumed as background knowledge in the hypothesis. This hypothesis is independent of the order of the cars and also independent of the size of the trains.

This example has shown the power of representation of first-order languages. The rich expressiveness was the motivation for many researchers to apply first-order logic language in machine learning. The work of these researchers created a new area in machine learning named Inductive Logic Programming (ILP).

3.2 Inductive Logic Programming

ILP is centred on the definition of new algorithms and on the establishment of a formal framework for the learning of relational descriptions in the form of logic programs. ILP can also be seen as an extension of the Logic Programing (LP) field toward the process of induction.

From a practical point of view, ILP has the purpose of developing techniques, algorithms, and ultimately software, for the induction of hypotheses from a set of correct examples (positive evidence) and, optionally, from a set of wrong examples (negative evidence), possibly employing known knowledge (background knowledge). The ILP problem can be defined formally as the inference of a theory P from an evidence E (which may be only positive E^+ or both positive and negative (E^+,E^-)) possibly using a background knowledge theory B, such that the following conditions hold:

1. $B \not\models E^+$ (*prior necessity*)
2. $\forall\ e^- \in E^- : B \not\models e^-$ (*prior satisfiability*)
3. $B \cup P \models E^+$ (*posterior sufficiency*)
4. $\forall\ e^- \in E^- : B \cup P \not\models e^-$ (*posterior satisfiability*)

From a theoretical point of view, ILP can be considered the dual to LP, while ILP describes the process of inducing logic programs from logical facts introduced by the user, LP is centred on the deduction of facts from a logic program introduced by the user. This approach produces a simple but productive view, induction can be seen as the inverse process of deduction. In some cases, it is possible to invert the deductive inference rules, thus obtaining new inductive inference rules. In this way, if $B \cup P \models E^+$, deriving E^+ by applying B and P is considered deduction, and deriving P from B and E^+ is induction.

In ILP, inductive operators have been developed by inverting well formalised deductive rules, namely, unification, resolution, and implication. One first common approach to learning from positive data is that of taking relative least general generalisations (*rlggs*) of clauses. Plotkin [Plotkin, 1971b,a] was the first to rigorously analyse the notion of generalisation as an automatisation of the process of inductive inference. He formalised the concept of least general generalisations (*lggs*) of atoms. The work was based on the importance of Robinson's unification to deduction, and searched for an analogue which would be useful in induction. In fact, when restricted to atoms with the same sign and predicate symbol, *lgg* is exactly the dual process to unification. *Lggs* were defined to be an inverse to unification.

Plotkin also extended the investigation to clauses. Since the use of the simple *lgg* of a set of clauses can be very restrictive for ILP, he defined the *lgg* of two clauses relative to clausal background knowledge, or *rlgg*.

Following the same idea, there have been some approaches about the inversion of the deductive rule $\models$. A general method for automated deduction is the *resolution rule* introduced in [Robinson, 1965] by Robinson. It is a complete and sound inference rule in first-order logic. Muggleton and Buntine introduced the inversion of resolution in [Muggleton and Buntine, 1992]. This inverse operator constitutes the base of the algorithm employed in the systems Cigol [Muggleton and Buntine, 1992] and Crustacean [Aha *et al.*, 1994].

The system Golem [Muggleton and Feng, 1992] uses inverse resolution jointly with least general generalisation to induce a set of Horn clauses from a set of positive examples, restricted to ground facts and background knowledge given in the form of extensionally defined predicates (ground facts).

Although inverse resolution is a powerful method for generating hypotheses, in practice it can easily lead to a combinatorial explosion of candidates. An alternative approach is to use inverse entailment to produce just the single most specific hypothesis that, together with the background knowledge, entails the evidence. Then, this most specific hypothesis can be used to bound a specific-to-general search through the hypothesis space, but with the restriction that only the hypotheses more general than this bound are considered. This approach is the basis of the algorithm that implements Progol [Muggleton, 2000, 1995].

A different approach to ILP that is not based on the inversion of a deductive operator was introduced by Quinlan in the system FOIL [Cameron-Jones and Quinlan, 1994; Quinlan and Cameron-Jones, 1995; Quinlan, 1996c]. This system generates programs as sets of first-order rules, and can be considered as the extension of some propositional learning approaches to first-order representation. Formally, the programs learned by FOIL are sets of first-order rules, where each rule is similar to a Horn clause but with two differences: first, the literals are not permitted to contain function symbols (this reduces the hypothesis search space); second, the literals appearing in the body of the rule can be negated. The learning algorithm employed by FOIL is based on traditional concept-learning techniques. In particular, it relates to the greedy Top-Down Induction of Decision Trees (TDIDT) algorithms [Breiman *et al.*, 1984] and the AQ family of algorithms [Michalski, 1969; Michalski *et al.*, 1986]. The method can be summarised as follows: to learn each rule, a general-to-specific search is performed, at each step adding a single new literal to the rule preconditions. The new literal may refer to variables already mentioned in the rule preconditions o postconditions, and may introduce new variables as well. At each step, it uses a *Foil-Gain* function to select one from the literal candidates. The algorithm can deal with noisy data; in this case, the search is continued until some trade-off occurs between rule accuracy, and complexity.

FOIL can be considered the first system based on first-order learning that really works efficiently on a broad range of problems involving large and noisy datasets. On other hand, Progol is currently one of the most employed systems in the ILP field, due to its high expressiveness. This system can address problems with a very complex structure such as proteins or molecules.

Other approach to relational learning is Linus [Lavrac *et al.*, 1991; Lavrac and Dzeroski, 1994]. The idea of this approach is to convert a restricted set of relational problems into propositional problems. Then, it applies propositional systems like decision tree learners or rule learners to learn a model. And finally, the propositional model is reconverted into a first-order model. In order to perform the conversion between relational and propositional languages, Linus has some important restrictions: the training examples are expressed as ground facts, but these can not contain recursive terms. The hypothesis language does not support recursive predicate definitions. Background knowledge is allowed, but also in a restrictive form.

Recently, [Blockeel and Raedt, 1998; Blockeel, 1999] have presented the system TILDE. The aim of this system is to extend top-down induction of decision trees to richer representation languages. In this case, they employ a flat representation of Prolog as FOIL do. They can deal with background knowledge (Prolog programs), and it works also for regression.

3.3 Applications

ILP systems have been employed in a wide range of applications. The main advantage of ILP with respect to other ML approaches such as Artificial Neural Networks (ANN) [Lau, 1992], or Support Vector Machines (SVM) [Cristianini and Schöllkopf, 2000] is the capability of capturing complex problems in a comprehensible way. This makes ILP especially appropriate for scientific theory formation tasks where the data are structures, the model may be complex, and the comprehensibility of the generated knowledge is essential. Some of the domains where ILP has been applied are [Bratko *et al.*, 1998]: drug design for pharmaceutical purposes [King *et al.*, 1992], natural language [Mooney and Califf, 1995], the learning of medical rules [Mizoguchi *et al.*, 1996], protein primary-secondary shape prediction [Muggleton *et al.*, 1992], and mutagenicity prediction [Srinivasan *et al.*, 1994].

Due to the direct relationship between ILP and LP, the inductive techniques have also been successfully applied to many phases of software engineering: logic program synthesis and transformation, for the induction of properties of programs or databases [Cohen, 1994], and for program testing and debugging [Shapiro, 1983; Mooney and Richards, 1992; Hernández and Ramírez, 2001].

A recent and brand new area where ILP is being successfully applied is data mining. This area is called Multi-Relational Data Mining [Dzeroski and Lavrac, 2001], and aims at developing practical algorithms for learning relational descriptions from relational databases in a declarative knowledge representation framework. Discovering data which consists of complex structured objects also falls within the scope of this area.

Classical approaches for data mining are propositional (statistical methods, neural networks, decision trees, ...) [Weiss and Indurkhya, 1998], in the sense that the rules or the knowledge that these systems are able to obtain can only refer to one row at a time. Moreover, these methods usually work with only one relation (table). However, most databases have more than one table and many interesting rules which are to be learned are non-propositional. Consequently, there is a need for pre-processing and transforming the data in order to de-normalise them and put them in a proper way in order to use propositional learning.

One classical example of this situation is the supermarket basket analysis. Usually, the database is de-normalised and transformed into a two dimensional matrix where rows enumerate all the baskets in a week or month and columns enumerate all the products. Consequently, each position in the matrix indicates whether the product was included in that basket or not.

The problem of this approach is that transformations usually require part of the knowledge to be discovered. The search is then more focussed towards a goal, and the knowledge that can be extracted is more expectable, and, consequently, less novel.

The potential users of these tools are usually known as 'farmers' [R. Kimball and ThornthWaite, 1998] rather than 'miners' or 'explorers'.

On the contrary, ILP approaches to data mining [Morik, 1997; Wrobel, 2001; Dzeroski and Lavrac, 2001] permit the generation of richer concepts, since the expressiveness of logic programming is universal. For instance, the following rule can be induced by an ILP system without any transformation at all of the original database:

```
good_client(X) :- person(X), married(X,Y), good_client(Y).
```

Even if we de-normalise into `person(X,M)` where M represents X's spouse, the resulting rule would still be relational, because it requires to compare two rows in a relationship:

```
good_client(X) :- person(X,Y), good_client(Y).
```

and, hence, not obtained by propositional learners. Moreover, ILP approaches excel statistical and artificial neural networks approaches in the fact that rules are much more understandable than in these approaches. Furthermore, background knowledge can be easily expressed manually and then exploited by the ILP system.

3.4 Representational Limitations

First-order languages have represented an important step beyond in the use of very expressive languages for learning. However, this kind of languages also has important limitations:

- **Functions**: functions are artificially expressed in first-order languages by using predicates. If the function has only two classes it is usually represented by predicates, using the examples of one class like positive examples, and the examples of the other class like negative example. When there are more than two classes, there is only positive evidence, and each example is expressed like a fact. In these cases, the attributes and the class are only distinguished through the use of modes. This representation based on modes and facts is not supported by many ILP systems. Let us consider, for instance, a classification problem originally appeared in [Cendrowska, 1987] about fitting contact lenses. In this dataset, a fact as `lens(young, myopia, astigmatism, normal, hard)` means that the contact lenses of a young person who has myopia and astigmatism, and produces a normal rate of tears, must be hard. Therefore, this last argument represents the class, the output of the classification problem. A more natural representation for this problem would be `lens(young, myopia, astigmatism, normal)=hard`.

- **Types**: First-order languages have a limited use of types. Constructor types cannot be defined naturally. The use of complex structures like sets and multisets is not direct.

- **Higher order**: Higher order is not supported by the basic first order representation. This is a drawback for the learning of complex problems where higher order features can be very useful. Instead, some ILP systems are schema-based, where these schemata cannot be learned.

- **Cost-sensitive learning**: Due to the artificial representation of functions, the definition of ILP cost-sensitive learning algorithms has not been very profuse, and it is unclear how this could be done for more than two classes and still be consistent with the truth/false basis of logic programming.

- **Numerical attributes**: The use of numerical types in logic languages is not direct. Hence the learning of problems that contains numerical attributes is not possible for most of the ILP systems unless they use some instance of constraint logic programming (CLP) scheme [Jaffar and Lassez, 1987].

Chapter 4

FLP and Knowledge Representation

In recent years, several proposals have been made to amalgamate functional and logic programming languages. These multi-paradigm languages combine features from functional programming (nested expressions, lazy evaluation), logic programming (logical variables, partial data structures), and concurrent programming (concurrent evaluation of constraints with synchronisation on logical variables).

From all the approaches to unify functional and logic programming in one paradigm, those that use narrowing [Reddy, 1985a] as operational semantics have monopolised the interest of great part of researchers. This is mainly because of the good properties on soundness and completeness of this operator. Concretely, a version of narrowing, *Needed Narrowing* [Antoy *et al.*, 2000] is the basis of the language Curry [Hanus , ed.]. The development of Curry is an international initiative intended to provide a common platform for the research, teaching and application of integrated functional logic languages.

In this chapter, we summarise some basic notions of the functional logic programming paradigm. First, we introduce some concepts and notations and provide a precise terminology. Then, we briefly present some notions and results regarding term rewriting systems. Finally, we conclude the chapter with an introduction to functional logic programming for knowledge representation, and with a discussion on the applicability of this kind of languages in machine learning.

4.1 Basic notions

We briefly review some basic concepts about equations, Term Rewriting Systems and $\mathcal{E}$-unification. For any concept which is not explicitly defined, the reader may refer to [Hanus, 1994; Klop, 1992; Siekmann, 1984]. Let Σ be a set of *function symbols* (or functors) together with their arity[1] and let $\mathcal{X}$ be a countably infinite set of *variables*. We consider Σ as the disjoint union $\Sigma = \mathcal{C} \uplus \mathcal{F}$ of symbols $c \in \mathcal{C}$, called *constructors*, and symbols $f \in \mathcal{F}$, called *defined functions*. Then $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of *terms* built from Σ and $\mathcal{X}$, and $\mathcal{T}(\mathcal{C}, \mathcal{X})$ is the set of constructor terms. $\mathcal{T}(\Sigma)$ represents the set of ground terms, and $\mathcal{T}(\mathcal{C})$ the set of ground constructor terms. The set of variables occurring in a term t is denoted $Var(t)$. This notation naturally extends to other syntactic objects (like clause, literal, ...). A term t is a *ground term* if $Var(t) = \varnothing$.

A *substitution* is defined as a mapping from the set of variables $\mathcal{X}$ into the set of terms $\mathcal{T}(\Sigma, \mathcal{X})$. Given a substitution θ and a set of variables $V \subseteq \mathcal{X}$, we denote by $\theta_{\upharpoonright V}$ the substitution obtained from θ by restricting its domain to V. We write $\theta = \sigma\ [V]$ if $\theta_{\upharpoonright V} = \sigma_{\upharpoonright V}$, and $\theta \leq \sigma\ [V]$ denotes the existence of a substitution γ such that $\gamma \circ \theta = \sigma\ [V]$. A term t' is an *instance* of t if there is a substitution σ with $t' = \sigma(t)$. This implies a *subsumption ordering* on terms which is defined by $t \leq t'$ iff t' is an instance of t. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. A unifier σ is the *most general* unifier (mgu) if $\sigma \leq \sigma'\ [\mathcal{X}]$ for each other unifier σ'.

By considering the usual representation of a term by a labelled tree, the *occurrences* are sequences of natural numbers denoting an access path in a term. Λ denotes the empty sequence. $O(t)$ and $\bar{O}(t)$ denote the *set of occurrences* and *non-variable occurrences* of t, respectively. $t_{|u}$ denotes the *subterm* of t at the occurrence u and $t[t']_u$ denotes the *replacement* of the subterm of t at the occurrence u by the term t'.

An equation is an expression of the form $l = r$ where l and r are terms. l is called the left hand side (lhs) of the equation and r is the right hand side (rhs). Given an equation $l = r$, a variable $x \in Var(r)$ such that $x \notin Var(l)$ is called an *extra variable*. An equational theory $\mathcal{E}$ (which we call *program*) is a finite set of equational clauses of the form $l = r \Leftarrow e_1, \ldots, e_n$ with $n \geq 0$ where e_i is an equation, $1 \leq i \leq n$. The theory (and the clauses) are called *conditional* if $n > 0$ and *unconditional* if $n = 0$. Given an equational theory $\mathcal{E}$, $=_\mathcal{E}$ denotes the finest congruence over $\mathcal{T}(\Sigma, \mathcal{X})$ containing all the pairs $(\sigma(s), \sigma(s'))$ for $s = s'$ in $\mathcal{E}$ and σ an arbitrary substitution. $\lfloor s \rfloor$ denotes the set of all terms t such that $s =_\mathcal{E} t$. An equational theory can also be viewed as a (Conditional) Term Rewriting System, (C)TRS, since the equation in the head is implicitly oriented from left to right and the literals e_i in the body

[1] We assume that Σ contains at least one constant.

are ordinary non-oriented equations. Given a (C)TRS $\mathcal{R}$, $t \to_{\mathcal{R}} s$ is a rewrite step if there exists an occurrence u of t, a clause $l = r \in \mathcal{R}$ and a substitution θ with $t_{|u} = \theta(l)$ and $s = t[\theta(r)]_u$. A term t is said to be in *normal form* w.r.t. $\mathcal{R}$ if there is no term t' with $t \to_{\mathcal{R}} t'$. We say that an equation $t = s$ is normalised w.r.t $\mathcal{R}$ if t and s are in normal form. Given a (C)TRS $\mathcal{R}$, $\mathtt{nf}_R(t)$ denotes the set of normal forms of a term t w.r.t. R, and $\mathtt{cnf}_R(t) = \mathtt{nf}_R(t) \cap \mathcal{T}(\mathcal{C}, \mathcal{X})$ is the set of constructor normal forms of t. $\to^*_{\mathcal{R}}$ is the transitive closure of $\to_{\mathcal{R}}$. $\mathcal{R}$ is said to be *canonical* if the binary one-step rewriting relation $\to_{\mathcal{R}}$ is terminating (there is no infinite chain $s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} s_3 \to_{\mathcal{R}} \ldots$) and confluent ($\forall\, s_1, s_2, s_3 \in \mathcal{T}(\Sigma, \mathcal{X})$ such that $s_1 \to^*_{\mathcal{R}} s_2$ and $s_1 \to^*_{\mathcal{R}} s_3, \exists\, s \in \mathcal{T}(\Sigma, \mathcal{X})$ such that $s_2 \to^*_{\mathcal{R}} s$ and $s_3 \to^*_{\mathcal{R}} s$). An $\mathcal{E}$-unification algorithm defines a procedure for solving an equation $t = s$ within the theory $\mathcal{E}$.

4.1.1 Narrowing

The operational semantics of functional logic languages is usually based on narrowing, a generalisation of the term rewriting relation which combines reduction and variable instantiation. Narrowing [Hussmann, 1986; Reddy, 1985b] is a sound and complete $\mathcal{E}$-unification method for theories which satisfy some requirements (such as confluence and termination properties or the absence of extra variables). To evaluate terms containing variables, narrowing non-deterministically instantiates the variables such that a rewrite step is possible (usually by computing most general unifiers between a subterm and the left-hand side of some rule [Hanus, 1994]). Formally, given a program P, a term t *narrows* into a term t' (in symbols $t \overset{u,l=r,\theta}{\hookrightarrow}_P t'$)[2] iff $u \in \bar{O}(t)$, $l = r$ is a new variant of a clause from P, $\theta = mgu(t_{|u}, l)$ and $t' = \theta(t[r]_u)$. We write $t \hookrightarrow^n_P t'$ if t narrows into t' in n narrowing steps. We denote by $t_0 \overset{\theta}{\hookrightarrow}{}^* t_n$ a sequence of narrowing steps $t_0 \overset{\theta_1}{\hookrightarrow} \ldots \overset{\theta_n}{\hookrightarrow} t_n$ with $\theta = \theta_n \circ \cdots \circ \theta_1$ ($\theta = id$ if $n = 0$). Since we are interested in computing *values* (constructor terms) as well as *answers* (substitutions) in functional logic programming, we say that the narrowing derivation $t \overset{\theta}{\hookrightarrow}{}^* c$ *computes the result* c *with answer* σ if c is a constructor term.

Narrowing derivations can be represented by a (possibly infinite) branching tree. A *failing leaf* contains an expression which is not a constructor term and which cannot be further narrowed. The following example extracted from [Hanus, 1994] shows the use of the narrowing method.

Example 1 *Consider the following program that defines the function* `append` *for con-*

[2]Or simply $t \overset{l=r,\theta}{\hookrightarrow}_P t'$ or $t \overset{\theta}{\hookrightarrow}_P t'$ if the occurrence or the clause is clear from the context. Also, the subscript P will usually be dropped when clear from the context.

catenating lists (the lists are represented with the Prolog *notation).*

$$\begin{array}{lcl} \texttt{append}([], \texttt{L}) & = & \texttt{L} \\ \texttt{append}([\texttt{E}|\texttt{R}], \texttt{L}) & = & [\texttt{E}|\texttt{append}(\texttt{R}, \texttt{L})] \end{array}$$

If we want to solve the equation `append(A,[2])=[1,2]`*, then we apply the second rule (instantiating* `A` *to* `[E|R]`*), and then the first* `append` *rule (instantiating* `R` *to* `[]`*):*

$$\begin{array}{lcl} \texttt{append}(\texttt{A}, [2]) = [1, 2] & \overset{\{\texttt{A}\backslash[\texttt{E}|\texttt{R}]\}}{\hookrightarrow} & [\texttt{E}|\texttt{append}(\texttt{R}, [2]) = [1, 2]] \\ & \overset{\{\texttt{R}\backslash[]\}}{\hookrightarrow} & [\texttt{E}, 2] = [1, 2] \end{array}$$

Some instantiations of clause variables in the substitution of narrowing derivations have not been shown for the sake of readability. The equation `[E,2]=[1,2]` *can be directly solved by unification which instantiates* `E` *to* `1`*. Thus, by composing the instantiations, narrowing returns the solution* $\{\texttt{A} \setminus \texttt{[1]}\}$*.*

A challenge in the design of functional logic languages is the definition of an appropriate strategy to solve equations. Such strategy should be at least *sound* (i.e., only correct solutions are computed) and *complete* (i.e., all solutions or more general representations of the solutions are computed). The narrowing relation can be used to define such a strategy. However, since simple narrowing can have a huge search space, great effort has been made in order to develop a narrowing strategy which limits this space (e.g., basic [Hullot, 1980], innermost [Fribourg, 1985], outermost [Echahed, 1990], standard [Middeldorp and Okui, 1998], lazy [Hans *et al.*, 1992], and needed [Antoy *et al.*, 2000]). Each strategy requires certain conditions on the rewrite system in order to ensure completeness. *Needed narrowing* is a strategy that, for *inductively sequential* programs, preserves the completeness of narrowing[3]. Moreover, it is currently the best known narrowing strategy due to its optimality properties w.r.t. the length of successful derivations and the number of computed solutions. The *Needed narrowing* strategy has been implemented in the Curry language.

Other different approaches to functional logic programming are based on *residuation*. Roughly speaking, residuation proceeds as follows: whenever an unbound variable needs to be bound in order to perform a computation, this particular computation is suspended and the execution waits for this variable to be bound by some other computation. When it is done, the suspended computation can continue. Escher [Lloyd, 1995], or Oz [Smolka, 1995] are programming languages based on residuation.

4.2 FLP for Knowledge Representation

The importance of the representation language in the learning process has been stated by different works [Bowers *et al.*, 2000; Flach, 2000]. An interesting discussion about

[3] A precise definition of this class of programs and the needed narrowing strategy is based on the notion of a *definitional tree* [Antoy, 1992].

this topic can be found in [Neri and Saitta, 1994]. In Section 2.4 we discussed about the limitations of propositional approaches for learning in complex scenarios, where the structure of each individual is expressed by a tuple of constants.

The use of logic programing for learning represents an important advance for the representation of complex individuals. However, this kind of languages also presents some drawbacks mainly due to the artificial treatment of types and functions (see Section 3.4).

As [Bowers *et al.*, 2002] stated "one can easily argue that, from a knowledge point of view, it would be far more desirable to capture directly the structures in the data with corresponding structures in its representation... Then, if we can design learning algorithms capable of manipulating such representations directly, it would be possible to apply machine learning more directly and naturally to domains rich in structure".

The main advantage of using a representation of examples that reflects directly their real structure, is that this representation can be used to guide the hypothesis construction. Types are essential to reach this direct mapping between real life and its machine interpretation. Functional logic languages like Curry [Hanus , ed.] or Escher [Lloyd, 1995] support these features.

By using Curry we can define the types to represent a train of the example seen in Section 2.4 as follows:

$$
\begin{array}{lcl}
\textit{type} Train & = & [Car] \\
\textit{type} Car & = & (CShape, CLength, CRoof, CWheels, Load) \\
\textit{data} CShape & = & Rect|Hexa|... \\
\textit{data} CLength & = & Long|Short \\
\textit{data} CRoof & = & None|Peak|... \\
\textit{type} CWheels & = & Int \\
\textit{type} Load & = & (LShape, LNumber) \\
\textit{data} LShape & = & Circ|Hexag|Recta|Tria|... \\
\textit{type} LNumber & = & Int
\end{array}
$$

This type signature implements a train by means of lists, tuples, and simple enumerate types. Taking into account this type definition, the first example is represented as follows:

$$
\textit{eastbound} \; :: \; Train -> \; Bool
$$

$$
\begin{array}{ll}
\textit{eastbound} & ([(Rect, Short, None, 2, (Circ, 1)), \\
 & (Rect, Long, None, 3, (Hexag, 1)), \\
 & (Rect, Short, Peak, 2, (Tria, 1)), \\
 & (Rect, Long, None, 2, (Recta, 3))]) \quad = \quad True
\end{array}
$$

Note, that here the examples are equations instead of facts. As you can see, functions are a more natural way to represent classification problems. Finally, the hypothesis is:

$$
\begin{aligned}
\mathit{eastbound}\ t \quad & \mid \mathit{member}\ x\ t\ =:=\ \mathit{True} \\
& \&\ \mathit{proy}2\ x\ =:=\ \mathit{Short} \\
& \&\ ((\mathit{proy}3\ x) == \mathit{None})\ =:=\ \mathit{False}\ =\ \mathit{True}\ \mathit{where}\ x\ \mathit{free}
\end{aligned}
$$

This function employs the logic programming features of a functional logic language, in this case Curry. The function *member* returns the components of a list, *proy*2 and *proy*3 returns the second and third components of a 5-tuple, respectively. The function *eastbound* returns *True* if there is a car in the train that is short, and it is not closed.

We have employed the same example in three different frameworks: propositional, logic programming, and functional logic programming. It has given a perspective of the drawbacks and limitations of each approach. In the propositional case, the examples were flattened into a fixed number of attributes, some of them empty in many cases. First-order languages introduce a very rich expressiveness that permits the use of lists, tuples, and background knowledge. Finally, functional logic languages increase the first order expressiveness by including functions, sets and multisets.

4.2.1 Learning and Functional Logic Programming

There are some previous works on the learning of functions and the induction of functional programs following different approaches such as in [Olsson, 1995; Rouveirol, 1994; Smith, 1984; Togashi and Noguchi, 1990]. The induction of functional logic programs has been discussed in [Bergadano and Gunetti, 1993], but it must not be confused to genuine functional-logic learning, because it is just a restricted variant of logic programs such that each n-ary predicate can be associated to a total function.

In [Quinlan, 1996d], the FFOIL system is described. FFOIL is a version of FOIL that is able to learn functions. In this system, a function with n arguments (the input variables) is represented by a $n+1$-ary relation where the additional argument represents the function outcome. The result obtained by FFOIL consists of a Prolog program with clauses of the learned relation and modes.

In addition, a framework for the induction of functional logic programs in Escher [Lloyd, 1995] has been presented in [Bowers *et al.*, 1997, 2000]. Escher is an integrated logic and functional programming language based on the Church theory of types which incorporates some higher-order concepts. The syntax of programs is functional (as in the Haskell language) and the computational model of Escher is based on the residuation mechanism. Since functions operate on data types with several data constructors, the proposed algorithm chooses one of the arguments as pattern for the induction of a function, and partitions the examples according to the constructor appearing in them in this argument. Then, one statement is learned for each case. this approach Therefore, this approach is based on pattern schemes.

Regarding higher order features, [Muggleton and Page, 1994] defends the importance of higher order logic for learning. [Bowers *et al.*, 2002] have proposed a powerful knowledge representation framework for inductive learning in higher order. First, they present the framework where it is possible to represent complex structures; this framework is based on higher order. They show that the structures employed for the representation can be extremely helpful for the guidance of the learning process. This paper also contains an interesting discussion with many references about representation languages in machine learning.

Chapter 5

Evaluation of Models

In this chapter, we review the techniques for evaluating the quality of models produced by machine learning systems. We only consider classification models.

First, we introduce the classical evaluation procedures for classifiers based on the accuracy of the model, i.e., the percentage of examples that the models are able to predict correctly.

Secondly, we address the issue of evaluating models according to its misclassification costs and test costs in a broad cost-sensitive context. In this way, we describe the goal of cost-sensitive learning.

5.1 Hypothesis Evaluation based on Accuracy

Mitchell [Mitchell, 1997] formalised, the goal of predictive machine learning as the learning of a target function f by considering a space H of possible hypotheses. To do this, the learning algorithm employ an evidence formed by training instances according to a distribution $\mathcal{D}$. This distribution defines the probability of encountering each instance in the space of all possible instances. Since it is possible to induce many and diverse hypotheses from the same evidence by using machine learning techniques, it is important to establish measures to rank the hypotheses according to its "distance" to f in order to know which is the best.

Usually, these measures compute the accuracy, or equivalently, the error rate of each model with respect to f. The error can be computed either over the sample of data that is available (*sample error*), or over the whole distribution $\mathcal{D}$ of examples (*true error*).

Definition 1 (Sample error.) *The sample error ($error_S(h)$) of a hypothesis h with* *respect to a target function f and a sample S is*

$$error_S(h) = \frac{1}{n}\sum_{x \in S} \delta(f(x) \neq h(x))$$

where n is the number of components of S, $\delta(true) = 1$, and $\delta(false) = 0$.

Definition 2 (True error.) *The true error ($error_T(h)$) of a hypothesis h with respect to a target function f and a distribution $\mathcal{D}$ is the probability that h will misclassify an instance randomly drawn according to distribution $\mathcal{D}$*

$$error_T(h) = Pr_{x \in \mathcal{D}}[(f(x) \neq h(x))]$$

where $Pr_{x \in \mathcal{D}}$ denotes that the probability is taken over the distribution $\mathcal{D}$.

We would like, of course, to know the true error. However, the sample error is all what is possible to compute, since the target function f is not known (except a partial extract from the training example).

A simple way to evaluate hypotheses is to learn them from the entire existing evidence and then use the same evidence to measure the sample error of each hypothesis. This simple method is not a good procedure, because it benefits the methods that overfit (see Section 2.2.1) the evidence, i.e. the hypotheses that are too centred on the evidence.

A better mechanism to measure the sample error ($error_S(h)$) of a hypothesis with respect to an evidence is to randomly separate the available evidence into two disjoint different subsets. The first one (called *training dataset*) is employed to learn the hypothesis, and the second one (called *test dataset*) is only used to compute the sample error. This procedure overcomes partially the overfitting problem of the previous evaluation procedure; however, the result is very dependent on the split performed to the dataset. Since the split of the original dataset is repeated randomly, two different experiments performed over the same hypothesis and sample could produce very different results.

One way to reduce this risk is to repeatedly divide the available dataset into k disjoint subsets. Then, a hypothesis is learned by using a dataset formed by the union of $k-1$ subsets, and the remaining dataset is employed to compute the partial sample error. This procedure is done k times, employing always a different subset to compute the partial sample error. The final sample error is computed as the average of the k partial sample errors. In this way, the learning algorithm is tested on k independent test sets. This method of evaluation is called *k-fold cross validation*.

5.1.1 Priors, Bias and Bayes

Given a hypothesis space H, and an observed training data D, we are interested in selecting which is the best hypothesis h from that space. One way to select this hypothesis is to consider the best hypothesis as the most probable hypothesis, given the data D plus any initial knowledge about the prior probabilities of the hypotheses in H.

Bayes theorem provides a theoretical formalism to compute such probabilities. Let us consider $P(h)$ as the initial probability that hypothesis h holds, before the training data has been observed. $P(h)$ is known as the *prior probability* of h. If that probability is not known, then we can assign the same value to each candidate. $P(D)$ denotes the prior probability that the training data D will be observed. Next, $P(D|h)$ denotes the probability of observing data D, when hypothesis h holds. Finally, $P(h|D)$ is the probability that h holds given the observed data D. $P(h|D)$ is called the *posterior probability*. In machine learning, we are interested in the posterior probability, because it reflects the confidence that h holds, depending on the training data D.

By employing the Bayes theorem, we are able to compute the posterior probability $P(h|D)$, from the prior probability $P(h)$, with $P(D)$ and $P(D|h)$:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Bayes theorem has also been employed for learning purposes, inside the family of Bayesian learning algorithms [Duda and Hart, 1974; Casella and Berger, 1990]. These algorithms are based on the assumption that optimal decisions can be made by reasoning about the probability features of the observed data.

In many cases, the goal is to select the most probable hypothesis h (or one of the maximally probable) from a set of candidate hypotheses in H, given the observed data D. Any such maximally probable hypothesis is named *maximum a posteriori* (MAP) hypothesis. We can select the MAP hypotheses by computing the posterior probability with the Bayes theorem. Formally, h_{MAP} is a MAP hypothesis if :

$$\begin{aligned} h_{MAP} &\equiv \mathtt{argmax}_{h \in H} P(h|D) \\ &= \mathtt{argmax}_{h \in H} \frac{P(D|h)P(h)}{P(D)} \\ &= \mathtt{argmax}_{h \in H} P(D|h)P(h) \end{aligned}$$

Note that we can drop $P(D)$ because it is a constant independent of h.

We have to approximate $P(h)$ in order to compute h_{MAP}. If we assume that H is finite and known, then we can consider the uniform distribution for $P(h)$.

$$P(h) = \frac{1}{|H|}$$

Otherwise, if H is infinite, then $P(h) \approx 0$. In either case, we can employ the *maximum likelihood*, defined as:

$$h_{ML} = \mathtt{argmax}_{h \in H} P(D|h)$$

However, following Occam's Razor principle, shorter hypotheses can be considered to be more probable. Occam's Razor is a principle attributed to the 14th century logician William of Ockham: "Entities should not be multiplied unnecessarily". This is nowadays understood to mean "Of two competing theories or explanations, all other things being equal, the simpler one is to be preferred" [Wikipedia, 2003]. The problem to formalise Occam's razor was to choose an objective measure of simplicity. Kolgomorov complexity [Li and Vitányi, 1997] is such a measure. Then, the probability of a hypothesis can be formalised as : $P(h) = 2^{-K(h)}$ where $K(\cdot)$ is the descriptional (Kolmogorov) complexity of h.

The problem of this approach is that D is not taken into account. We can choose a different $P(h)$ useful for cases where H is infinite. There exists a suitable paradigm for computing the complexity of the hypothesis: the *Minimum Description Length* (MDL) principle. If we assume $P(h) = 2^{-K(h)}$ and $P(D|h) = 2^{-K(D|h)}$, we can obtain the (MAP) hypothesis as follows [Li and Vitányi, 1997]:

$$
\begin{aligned}
h_{MAP} &= \texttt{argmax}_{h \in H} P(h|D) \\
&= \texttt{argmax}_{h \in H} \frac{P(D|h)P(h)}{P(D)} \\
&= \texttt{argmax}_{h \in H} P(D|h)P(h) \\
&= \texttt{argmax}_{h \in H} \log P(D|h)P(h) \\
&= \texttt{argmax}_{h \in H} \log 2^{-K(D|h)} + \log 2^{-K(h)} \\
&= \texttt{argmax}_{h \in H} (-K(D|h) - K(h)) \\
&= \texttt{argmin}_{h \in H} (K(h) + K(D|h))
\end{aligned}
$$

This last expression is the MDL principle, which means that the best hypothesis is the one which minimises the sum of the description of the hypothesis and the description of the evidence using the hypothesis.

5.2 Cost-sensitive Learning

In general, classifiers are used to make predictions for decision support. Since predictions can be wrong, it is important to know what the effect is when the predictions are incorrect. In many situations not every error has the same consequences. Some errors can have greater cost than others, especially in diagnosis. For instance, a wrong diagnosis or treatment can have different cost and dangers depending on which kind of mistake has been done. In fact, it is usually the case that misclassifications of minority classes into majority classes (e.g. predicting that a system is safe when it is not) have greater costs than misclassifications of majority classes into minority classes (e.g. predicting that a system is not safe when it actually is). Obviously, the costs of each misclassification are problem dependent, but it is almost never the case that they would be uniform for a single problem. Consequently, accuracy is not generally the best way to evaluate the quality of a classifier or a learning algorithm.

Cost-sensitive learning is a more realistic generalisation of predictive learning, and cost-sensitive models allow for a better and wiser decision making. In this way, the quality of a model is measured in terms of cost minimisation rather than in terms of error minimisation.

Consider the following example: the emergency room of a hospital is determined to refine its admission decision chart using machine learning techniques. Whenever a new patient arrives to the emergency room, after the initial care and evaluation, three possible destinations are possible depending on the patient comorbid conditions: discharge, ward and ICU. The percentage of cases in each of these three classes is 86,5%, 13% and 0,5%, respectively. If we gather examples of previous patients and pass them through a machine learning algorithm, a model that always discharges could be obtained, because it would have a precision of 86,5%, and, in terms of accuracy, it is a good model. However, this model is useless. Even more important, it is quite dangerous, since patients with comorbid conditions would be sent home, which is plainly a disastrous policy.

The idea is to obtain the model that minimises the classification cost, rather than the model that minimises the classification error. Consider that we are able to evaluate the cost of each misclassification. This could be expressed in terms of a misclassification cost matrix:

		Actual		
		Discharge	Ward	ICU
Predicted	Discharge	0	5,000	500,000
	Ward	300	0	50,000
	ICU	800	500	0

If we evaluate classifiers with their misclassification cost, it is obvious that the classifier that always predicts *discharge* (86,5% accuracy) would have a cost of 90,000 cost units/100 patients, the classifier that always predicts *ward* (13% accuracy) would have a cost of 26,200 cost units/100 patients, and the classifier that always predicts *ICU* (0,5% accuracy) would have a cost of 75,700 cost units/100 patients. We can do much better than that if we derive classifiers that distribute the errors in such a way that the total cost is minimised. This is a first approach to misclassification cost reduction, and has been addressed with diverse success by several approaches, as we discuss later (Section 5.2.3).

Other important kind of costs are the costs associated with testing the attributes, i.e. to evaluate the condition on the input variables of the classification problem. In application areas such as medicine, some tests are very expensive, or even painful for the patient. A model that uses fewer or less costly attributes is preferable over a model that uses many expensive attributes. Therefore, it is sometimes essential to find hypotheses that minimise the test costs, without losing too much accuracy.

There are other different kinds of cost associated with predictions (see e.g [Turney, 2001]), although they are less relevant than misclassification and test costs. In the remainder of this section, we concentrate on the learning methods that tend to minimise the misclassification costs and the costs associated with testing an attribute.

5.2.1 Basic Notions on Cost-sensitive Learning

In this section we include some elementary notions about cost-sensitive learning regarding misclassification costs.

Misclassification Results and the Confusion/Misclassification Matrix

Given a classifier, it is usual that its accuracy could be lower than 100%, let us say, for instance, 87,5%. In this case, it may be interesting to know to which class the misclassified 12,5% goes and how this error is distributed.

A confusion or contingency matrix is a very practical and intuitive way of seeing such a distribution. This matrix contains information about the distribution of the classes predicted with a classifier for a set of examples with respect to the correct classification.

Example 2 *Given 100 test examples and a classifier, an example of a confusion matrix for three classes $\{a, b, c\}$ might be:*

		Actual		
		a	b	c
	a	20	2	3
Predicted	b	0	30	3
	c	0	2	40

This matrix is understood as follows. From the hundred examples, 20 were of class a and all were correctly classified, 34 were of class b from which 30 were correctly classified as b, 2 misclassified as a and 2 misclassified as c. Finally, 46 examples were of class c from which 40 were correctly classified as c, 3 misclassified as a and 3 misclassified as b.

We use the notation M for this matrix and $M(i, j)$ refers to the element of predicted class i and actual class j.

Note that it is easy to obtain derived information from the confusion matrix. The vertical sums give the distribution of classes in the actual examples; the horizontal sums give the distribution of classes produced by the classifier.

A confusion ratio matrix can also be obtained by normalising each column to one.

Example 3 *The confusion ratio matrix corresponding to the matrix of example 2 would be:*

		Actual		
		a	b	c
	a	1	0.059	0.065
Predicted	b	0	0.882	0.065
	c	0	0.059	0.87

This matrix is represented by $\overline{M}$. Note that this matrix cannot be used instead of the original one because we have lost the class distribution.

As we have discussed before, frequently accuracy is a much too simplified measure of the quality of a classifier. For instance, given a dataset whose distribution of classes is ($p_a = 0.85, p_b = 0.1, p_c = 0.05$), i.e., most of the examples are of the class a, a simple classifier predicting everything into class a would have 85% of accuracy.

Apart from using confusion matrices to avoid this kind of oversimplification, which can be misleading for assessing the quality of classifiers, other measures have been introduced by using the separate predictive accuracies ($PAcc_i$). These can be based on the mean of the separate predictive accuracies, the product, the maximum or derived from informativeness measures. All of them try to balance the results of the accuracy of different classes, in order to improve the results for minority classes.

Cost matrix

Given a classification problem domain, the information about the cost of correct or wrong classifications can be given in very different degrees of detail.

One of the most complete way to provide the information about misclassification costs is a Cost Matrix (also known as Loss Matrix), which indicates the costs for correct and incorrect classifications.

Example 4 *A Cost Matrix for three classes $\{a, b, c\}$ might be as follows:*

		Actual		
		a	b	c
	a	-2.5	4	2
Predicted	b	2.1	-3.5	0
	c	1.2	1.3	-4

This example shows the usual portrait, the diagonal of the matrix shows the costs for correct classification $(-2.5, -3.5, -4)$. These values are usually negative or zero, because a correct classification has benefits instead of costs. The other values represent different cases of misclassification. For instance, the value 2.1 in cell (b, a) means that classifying incorrectly an a instance as a b instance has a cost of 2.1.

We use the notation C for this matrix and $C(i, j)$ refers to the element of predicted class i and actual class j.

From this matrix and the confusion matrix it is very easy to compute the cost of a classifier for a given dataset, just as the matrix product, given a Resulting Matrix R:

$$R(i,j) = M(i,j) \cdot C(i,j)$$

Just adding all the elements of the matrix, we have the overall cost:

$$Cost = \sum_i \sum_j R(i,j)$$

Example 5 *For the examples 2 and 4, the resulting matrix would be:*

		Actual		
		a	b	c
	a	-60	8	6
Predicted	b	0	-105	0
	c	0	2.6	-160

And the overall cost:

$$Cost = \sum_i \sum_j R(i,j) = -308.4$$

For small datasets or when the costs are computed for parts of a model (e.g. a part of a decision tree), it may be useful to compute a Laplace-corrected confusion matrix before, obtained as [Margineantu and Dietterich, 2000]:

$$M'(i,j) = n \cdot \frac{M(i,j) + \lambda}{c^2 \cdot \lambda + n}$$

where λ determines the strength of the Laplace correction, c the number of classes and n the cardinality of the dataset.

Cost Matrix Properties

Elkan [Elkan, 2001] presents some desirable cost matrix properties for two-class problems. Here we extend those properties to multi-class problems.

It is reasonable to assume that costs are greater for misclassification than for correct classification. This reasonableness condition can be stated as:

$$\forall i, j, i \neq j : C(i,i) < C(i,j)$$

The $<$ sign could be replaced by a $\leq$ to make this condition looser.

Another property that can be expected from a cost matrix is that there are no dominant rows, i.e., a row for which all the costs are lower than the corresponding costs of other row. More formally:

$$\neg\exists j, k : \forall i : C(k,i) < C(j,i)$$

If this condition would not hold, there will be some classes that will never be predicted in any situation whatsoever.

5.2.2 ROC Analysis

The usefulness of cost-sensitive learning does not only apply when the cost matrix is known a priori. If the cost matrix is not known or the distribution of the classes is unknown, one or many classifiers can be generated in order to behave well in the wider range of circumstances or contexts as possible. The Receiver Operating Characteristic (ROC) analysis [Egan, 1975; Swets, 1986; Provost and Fawcett, 1997; Swets *et al.*, 2000] provides tools to select a set of classifiers that would behave optimally and reject some other useless classifiers.

The ROC analysis makes it possible to evaluate a classifier performance in a more independent and complete way than just using accuracy. ROC analysis has usually been presented for two classes, because it is easy to define, to interpret and it is computationally feasible.

In ROC analysis with two classes the following notation is used for the confusion matrix:

		Actual	
		T	F
Predicted	T	True Positives (TP)	False Positives (FP)
	F	False Negatives (FN)	True Negatives (TN)

The ROC Analysis is performed using the confusion ratios, which are easily obtained:

- True Positive Rate: $TPR = TP/(TP + FN)$
- False Negative Rate: $FNR = FN/(TP + FN)$
- False Positive Rate: $FPR = FP/(FP + TN)$
- True Negative Rate: $TNR = TN/(FP + TN)$

This forms a confusion ratio matrix $\overline{M}$:

		Actual	
		T	F
Predicted	T	TPR	FPR
	F	FNR	TNR

Logically, $TPR = 1 - FNR$ and $FPR = 1 - TNR$. These latter equivalences provide a means to just selecting two of the ratios and constructing a bidimensional space with them, with a working area between (0,0) and (1,1).

ROC analysis is based on plotting the true-positive rate over the y-axis and the false-positive rate over the x-axis.

Example 6 *Given the following confusion matrix:*

		Actual	
		T	F
Predicted	T	30	30
	F	20	70

we would have a TPR= 0.6 and a FPR= 0.3. We can easily draw this point in the ROC space (also called Lorentz diagram [Hand, 1997]) as it is shown in Figure 5.1.

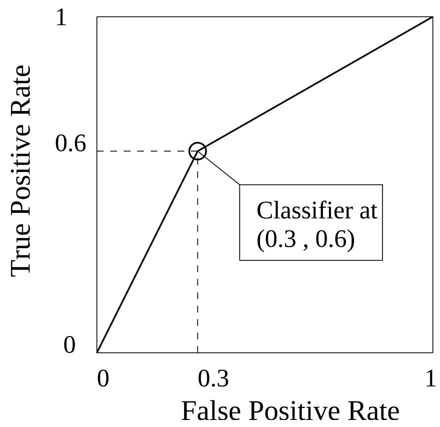


Figure 5.1: Representation of a classifier in the ROC space.

The points (0,0) and (1,1) represent, respectively, the classifier that classifies anything as negative and the classifier that classifies anything as positive. These are usually known as trivial classifiers.

There are many interesting results about ROC analysis (summarised from [Provost and Fawcett, 1997]):

- Once a test set is presented, the actual cost given by the classifier can be computed from the actual probabilities of the two classes ($p(T)$ and $p(F)$) and the cost matrix C, by:

  $$\texttt{Cost} = p(T) \cdot (FNR) \cdot C(F,T) + p(F) \cdot (FPR) \cdot C(T,F)$$

  From the previous formula, two classifiers 1 and 2 will have the same cost when:

  $$\frac{TPR_2 - TPR_1}{FPR_2 - FPR_1} = \frac{p(F) \cdot C(T,F)}{p(T) \cdot C(F,T)} = m$$

  This m can be seen as the slope of a line. All classifiers in the same line have the same performance and the lines are called *iso-performance* lines.

- Given two classifiers, we can obtain as many derived classifiers as we want all along the segment that connects them, just by voting them with different weights. Consequently, any point "below" that segment will have a greater cost for any class distribution and cost matrix, because it has lower true positive rate and higher false positive rate.

- According to the previous property, given several classifiers, one can discard the classifiers that fall under the convex hull formed by the points representing these classifiers. The convex hull that is formed with all the classifiers (jointly with the points (0,0), (1,0) and (1,1)) is known as ROC Curve.

- Due to the previous rationale, the best learning system is the one that produces a set of classifiers that maximises the Area Under the ROC Curve (AUC).

A typical snapshot of a ROC diagram is illustrated in Figure 5.2. Suppose that the five classifiers (plotted as A, B, C, D and E) have been obtained. Classifiers B and E fall under the ROC Curve (the convex hull formed by the five classifiers and the two trivial classifiers). Hence, B and E are discarded.

One problem of the classical ROC analysis is the difficulty to know the expected cost of a classifier once it has been selected as optimal for a given cost matrix. This problem has partially been solved by [Drummond and Holte, 2000]. They proposed a representation with same good properties of ROC representation based on the expected cost of the classifiers. In this representation, it is straightforward to reckon the expected cost of a classifier from its position in the space.

ROC Analysis for n classes

ROC analysis has been employed in problems with 2 classes, due to the good properties presented in the previous section. Srinivasan has shown [Srinivasan, 1999] that,

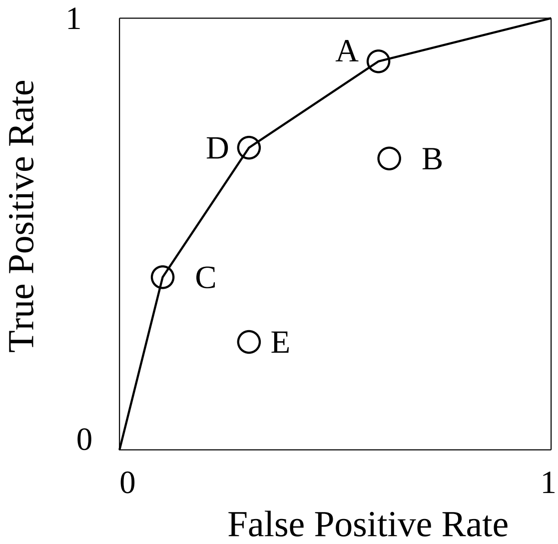


Figure 5.2: ROC curve for 5 classifiers.

theoretically, ROC analysis extends to more than two classes "directly". For c classes, and assuming a normalised cost matrix, we have to construct a vector of $d = c(c-1)$ dimensions for each classifier. In general the cost of a classifier for c classes is:

$$\texttt{Cost} = \sum_{i,j,i \neq j} p(i) \cdot C(i,j) \cdot \overline{M}(i,j)$$

where $\overline{M}$ is the confusion ratio matrix and $p(i)$ is the absolute frequency of class i. From the previous formula, two classifiers 1 and 2 will have the same cost when they are on the same iso-performance hyperplane. However, the $d-1$ values of the hyperplane are not so straightforward and easy to obtain and understand as the slope value of the bi-dimensional case.

Provided this direct theoretical extension, there are some practical problems.

- In two dimensions, doubling $p(T)$ (i.e. doubling the probability of one class) has the same effect on performance as doubling the cost $C(F,T)$ or halving the cost C(T,F). Is this still true for more than two dimensions? The answer is no in one direction, because for each change in the class distributions, there are infinite many corresponding cost matrices due to many more degrees of freedom.

- The best algorithm for the convex hull generation is $\mathcal{O}(N\log(N) + N^{d/2})$ [Lane, 2000; J.D. Boissonat, 1998], where N is the number of points and d the number of dimensions.

- In the 2-d case, it is relatively straightforward to know how to detect the trivial classifiers and the points for the minimum and maximum cases.

However, not only there are computational limitations but representational ones. ROC analysis in two dimensions has a very nice and understandable representation, but this cannot be directly extended to more than two classes, because even for 3 classes we have a 6-dimensional space, quite difficult to represent on a paper or a screen.

Since the computation of the ROC Curve in problems with more than two classes is very expensive, and it is even not clear how to be generated, the AUC value has not been frequently employed for the evaluation of this kind of problems. However, there exist some works that present simplified approximations that allows the AUC value to be computed for multi-class problems.

Hand and Till present a generalisation of the AUC measure in [Hand and Till, 2001]. It has been shown that for two dimensions the AUC measure is equivalent to the GINI measure (note that the GINI measure is not the GINI splitting criterion used in the CART algorithm, in this case it corresponds to the GINI coefficient [Gini, 1955]).

The idea is that the AUC measure for 2 dimensions can be defined by using the estimated probability that an example x_i pertains to class 0 (denoted by $p(\cdot)$ and estimated from the training set). This probability is used to rank the pairs $\{g_k, f_k\}$ where g_k and f_k are defined as $g = p_0(x1)$ and $f = p_0(x0)$ where $x1$ are the examples from the test set of class 1 and x_0 are the examples of class 0 from the test set.

The ranking is performed as described in [Hand and Till, 2001]. Let us denote with r_i the rank of the i^{th} class 0 test set point. Let us denote $S_0 = \sum r_i$. Then, they derive the area as

$$\hat{A} = \frac{S_0 - n_0(n_0+1)/2}{n_o n_1}$$

A relevant novelty of the paper is that they understand $\hat{A}$ as "an overall measure of how well separated are the estimated distributions of $p_0(\cdot)$ for class 0 and class 1", i.e., $\hat{A}(i,j)$ could be computed for whatever pair of classes i and j. This interpretation permits what they call "a simple generalisation of the AUC for multiple class classification problems". They define a new measure M as:

$$M = \frac{1}{c(c-1)} \sum_{i \neq j} \hat{A}(i,j) = \frac{2}{c(c-1)} \sum_{i<j} \hat{A}(i,j)$$

Fawcett has also proposed other approximation. This, presented in [Fawcett, 2001], is similar to Hand and Till's, but includes probabilities. Its definition is as follows:

$$AUCF = \sum_c p(i) \cdot \hat{A}(i, rest)$$

where $\hat{A}(i, rest)$ computes the AUC of one class against all the others, i.e., each time it considers all the other classes as one. This seems to be a simplification measure of Hand and Till's M function. The main difference lies in the fact that $AUCF$ includes the probability of each class as a factor and this might make $AUCF$ less independent of class probabilities and biased datasets. Fawcett defined this approach in [Fawcett, 2001] because he was not aware of the Hand and Till's measure[1].

Other interesting approximation can be found in [Mossman and Somoza, 1991]. In this case, the extension is just limited to 3 classes, and it is based on a priorisation of one of them. In this way, the problem is simplified to an approximate problem with as many dimensions as classes.

Finally, let us note that AUC is a better value than accuracy to establish the quality of classifiers, because AUC is more independent of the context (class distributions or misclassification costs) [Provost and Fawcett, 2001]. Therefore, AUC will replace more and more accuracy as a quality measure of classifiers. All the experiments of this dissertation will be presented in both values. Chapter 11 details how we compute the AUC measure in an efficient way.

5.2.3 Using Cost Information for Learning

In this section, we study some techniques which take into account the costs of classification errors in decision trees. Some similar works can be found in [Breiman *et al.*, 1984; Knoll *et al.*, 1994; Turney, 1995a].

Conceptually, the strategies of cost-sensitive learning are three [Margineantu, 2002]:

- Manipulating the training data.
- Making the learning algorithm cost-sensitive.
- Altering the outputs of the learned model.

The advantage of the first kind of strategies is that these methods are independent of the learning methods, because they only need to modify the training data, and then learn a model as usual. In decision tree learning, the second strategy is usually performed by modifying the splitting criterion, and the third one, by assigning the class of the leaves depending on its associated cost.

Manipulating the Training Data

A common approach to cost-sensitive learning is to modify the class distributions in the training dataset according to the misclassifications cost associated to the classes.

[1]Personal communication.

For two-class problems, this procedure is usually called *stratification* [Japkowicz, 2000], and it is performed by undersampling the examples of the class with less cost, or by oversampling the examples of the most expensive class. A generalisation of this procedure for multi-class problems has been defined by Margineantu in [Margineantu, 2002]. Another approach that employs training data manipulation is MetaCost [Domingos, 1999]. In this case the approach is based on relabelling training examples with their estimated minimal-cost classes, and then applying a non cost-sensitive learner to the new training dataset.

Splitting Criteria and Cost

The sensitivity of splitting criteria in decision tree learning has recently been studied experimentally and theoretically [Turney, 1995a; Drummond and Holte, 2000; Elkan, 2001]. Although some experimental studies have shown that changing the class distribution can be relatively effective [Weiss and Provost, 2001], some others have shown that usual splitting criteria (gain, gain ratio, gini and DKM) are insensitive in general to a distribution change [Drummond and Holte, 2000; Elkan, 2001].

According to the experimental analysis of [Drummond and Holte, 2000], the Accuracy (Expected Error, see Section 2.2.1) splitting criterion is the most sensitive to distribution changes, while the others are relative insensitive, and DKM being the most insensitive of all. The theoretical study performed by Elkan [Elkan, 2001] reinforces and even sharpens these experimental results: DKM is proved to be completely insensitive, and the effectiveness on C4.5 is usually small. In any case, the general conclusion is that changing the class distribution (even in the optimal way derived by [Elkan, 2001]) when any of these splitting criteria is used "is not likely to have much influence on the growing phase of decision tree learning".

Assigning the Class to the Leaves

Other method consists in the assignment of the class of the leaves according to a cost matrix, i.e., the class of a node will not be computed as the majority class, but as the class that minimises the expected cost of the examples that fall under that node using the current cost matrix.

If we define $Cost_i$ as the cost if class i would be selected:

$$Cost_i = \sum_j n(j) \cdot C(i,j)$$

where $n(j)$ is the number of training examples of class j, then the class is assigned as:

$$Class = \arg\min_i Cost_i$$

This way has been shown to be much more effective that the other two [Bradford *et al.*, 1998].

5.2.4 Test Costs

Apart from misclassification costs, there is another kind of costs that can be extremely important in some applications, especially in medical diagnosis. These are related to the cost of the tests that must be performed in order to make a prediction. Therefore, a model that performs a test with lower costs (in the wide sense of the term) is then preferable to a similar model that performs further and costlier tests. A usual way to express the information about the cost of the test is to use a vector of n components, where n is the number of attributes. Each component of the vector refers to the cost of performing the test of an attribute.

The following example illustrates the importance of taking into account the test cost when learning models:

Example 7 *Consider an imaginary diagnosis problem for three different diseases* Disease1, Disease2 and Disease3 *as follows:*

`BP − Min(MinimumBloodPressure)` :	*numerical.*	
`BP − Max(MaximumBloodPressure)` :	*numerical.*	
`Div_End(Diverticulities_through_Endoscopy)` :	*Nominal* :	*pos/neg.*
`Cysts_Scopy(CyststhroughColonoscopy)` :	*Nominal* :	*pos/neg*
`Meningitis_Lumbar(MeningitisthroughLumbarPuncture)` :	*Nominal* :	*pos/neg*
`Cysts_Echo(CyststhroughEchography)`	*Nominal* :	*pos/neg*
`Glucose_BA(GlucoseconcentrationthroughBloodAnalysis)` :	*numerical*	
`Leucocytoses_Urin(LeucocytosesthroughUrineAnalysis)` :	*Nominal* :	*pos/neg*

The test costs (taking into account economic, risk and pain issues) have been determined as follows:

`BP − Min` :	1 *cost units.*
`BP − Max` :	1 *cost units.*
`Div_End` :	30 *cost units.*
`Cysts_Scopy` :	50 *cost units.*
`Meningitis_Lumbar` :	200 *cost units.*
`Cysts_Echo` :	15 *cost units.*
`Glucose_BA` :	15 *cost units.*
`Leucocytoses_Urin` :	10 *cost units.*

And now consider that we have three decision trees for this problem:

Decision Tree 1:

$\texttt{Disease(BP} - \texttt{Min, BP} - \texttt{Max, Div\_End, Cysts\_Scopy, Meningitis\_Lumbar,}$
$\quad \texttt{Cysts\_Echo, Glucose\_BA, Leucocytoses\_Urin)}$ [class]
$\{DT1 - N1\}$: Cysts_Scopy = neg, Glucose_BA >= 120 [Disease1]
$\{DT1 - N2\}$: Cysts_Scopy = neg, Glucose_BA < 120, Leucocytoses_Urin = neg [Disease1]
$\{DT1 - N3\}$: Cysts_Scopy = neg, Glucose_BA < 120, Leucocytoses_Urin = pos [Disease2]
$\{DT1 - N4\}$: Cysts_Scopy = pos [Disease3]

Decision Tree 2:

$\texttt{Disease(BP} - \texttt{Min, BP} - \texttt{Max, Div\_End, Cysts\_Scopy, Meningitis\_Lumbar,}$
$\quad \texttt{Cysts\_Echo, Glucose\_BA, Leucocytoses\_Urin)}$ [class]
$\{DT2 - N1\}$: BP_Min >= 100, Cysts_Echo = pos [Disease3]
$\{DT2 - N2\}$: BP_Min >= 100, Cysts_Echo = neg, Meningitis_Lumbar = neg [Disease1]
$\{DT2 - N3\}$: BP_Min >= 100, Cysts_Echo = neg, Meningitis_Lumbar = pos [Disease2]
$\{DT2 - N4\}$: BP_Max >= 150, Cysts_Echo = pos [Disease3]
$\{DT2 - N5\}$: BP_Max >= 150, Cysts_Echo = neg, Meningitis_Lumbar = neg [Disease1]
$\{DT2 - N6\}$: BP_Max >= 150, Cysts_Echo = neg, Meningitis_Lumbar = pos [Disease2]
$\{DT2 - N7\}$: BP_Min < 100, BP_Max < 150, Div_End = pos [Disease3]
$\{DT2 - N8\}$: BP_Min < 100, BP_Max < 150, Div_End = neg [Disease1]

Decision Tree 3:

$\texttt{Disease(BP} - \texttt{Min, BP} - \texttt{Max, Div\_End, Cysts\_Scopy, Meningitis\_Lumbar,}$
$\quad \texttt{Cysts\_Echo, Glucose\_BA, Leucocytoses\_Urin)}$ [class]
$\{DT3 - N1\}$: Leucocytoses_Urin = neg, Cysts_Echo = pos [Disease3]
$\{DT3 - N2\}$: Leucocytoses_Urin = neg, Cysts_Echo = neg [Disease1]
$\{DT3 - N3\}$: Leucocytoses_Urin = pos [Disease2]

The three previous trees use different attributes to make a diagnosis. If we do not have any additional information apart from their accuracy we would have to select the most accurate one or the one with highest AUC.

However, let us consider that we have determined, by using e.g. the training set, what the frequency that an example falls into each node is:

Decision Tree 1:	Decision Tree 2:	Decision Tree 3:
{DT1-N1} (0.2)	{DT2-N1} (0.1)	{DT3-N1} (0.4)
{DT1-N2} (0.3)	{DT2-N2} (0.03)	{DT3-N2} (0.5)
{DT1-N3} (0.1)	{DT2-N3} (0.05)	{DT3-N3} (0.1)
{DT1-N4} (0.4)	{DT2-N4} (0.15)	
	{DT2-N5} (0.02)	
	{DT2-N6} (0.05)	
	{DT2-N7} (0.15)	
	{DT2-N8} (0.45)	

From the previous information we can compute the mean test cost of an example. The mean cost of a tree of n *leaves is performed in the following way :*

$$Mean\ Cost\ DT = \sum_{1 \le i \le n} (frequency(DT - Ni) \cdot \sum_{a \in Test_i} Cost(a))$$

where $Test_i$ represents the set of attributes that are tested for the leaf i.

Mean test cost for decision tree 1:
{DT1-N1} 0.2 · [50 + 15] = 12
{DT1-N2} 0.3 · [50 + 15 + 10] = 22.5
{DT1-N3} 0.1 · [50 + 15 + 10] = 7.5
{DT1-N4} 0.4 · [50] = 20
TOTAL: ***62 cost units****.*

Mean test cost for decision tree 2:
{DT2-N1} 0.1 · [1 + 15] = 1.6
{DT2-N2} 0.03 · [1 + 15 + 200] = 6.48
{DT2-N3} 0.05 · [1 + 15 + 200] = 10.8
{DT2-N4} 0.15 · [1 + 15] = 2.4
{DT2-N5} 0.03 · [1 + 15 + 200] = 6.48
{DT2-N6} 0.05 · [1 + 15 + 200] = 10.8
{DT2-N7} 0.15 · [1 + 1 + 30] = 4.8
{DT2-N8} 0.45 · [1 + 1 + 30] = 14.4
TOTAL: ***57.76 cost units****.*

Mean test cost for decision tree 3:
{DT3-N1} :- 0.4 · [10 + 15] = 10
{DT3-N2} :- 0.5 · [10 + 15] = 12.5
{DT3-N3} :- 0.1 · [10] = 1
TOTAL: ***23.5 cost units****.*

As we can see, the third decision tree has an average test cost quite lower than the other two trees, and in this regard, it is preferable over the rest. We can also see that the frequency of each node is very relevant. For instance, decision tree 2 employs the most expensive test (`Meningitis_Lumbar`). However, its use is quite infrequent and turns out to be, in overall terms, a decision tree which is less expensive than the first one. Note that a combination of the three trees, by using any fusion method, would have as cost the sum of all the single costs, i.e.: 143.26.

Provided that the three decision trees have similar accuracy and AUC, it is then much more preferable to use the third decision tree, because the test cost would be minimised. Consequently, economic, risk and pain issues are minimised.

It is also possible to take into account this cost in the learning process. For instance, in the case of decision trees, when constructing each tree, a splitting criterion is used to assign different degrees of optimality to each split. This optimality can be modified taking test-cost into account. In [Núñez, 1988], the author replaces the original *Gain* splitting criterion by the measure:

$$Gain'(S, A) = \frac{2^{Gain(S,A)}}{(Cost(A) + 1)^w}$$

where A is an attribute, S is the set of examples where the split is tested, $Cost(A)$ is the cost associated to testing attribute A, and w is a constant that determines the relative importance of cost versus information gain. He demonstrates that this approach obtain cheaper decision trees without sacrificing accuracy.

Other similar approach was proposed in [Tan and Schlimmer, 1990; Tan, 1993]. In this case the proposed modification is:

$$Gain'(S, A) = \frac{Gain^2(S, A)}{Cost(A)}$$

A comparison of these two approaches over different tasks can be found in [Núñez, 1991].

On other hand, [Turney, 1995a] introduces ICET, a hybrid decision tree induction algorithm, and use it to minimise the cost of classification where the tests are expensive. Since this is really an ILP system, we describe it below.

5.3 Cost-sensitive ILP

Due to the representational limitations of problems with more than 2 classes, there are not many works that adapt ILP techniques into a cost-sensitive context where there are misclassification costs. The existing approaches are mainly based on the application of 2-class ROC analysis for discriminating models.

For instance, a work that applies ROC analysis in first-order learning can be found in [Srinivasan *et al.*, 1999]. In the paper, an application of the ROCCH method [Provost and Fawcett, 1998, 2001] to evaluate models for toxicology prediction is shown.

Later, Srinivasan [Srinivasan, 2001] presented a method that incorporates ROC analysis in ILP in order to reject models that cannot be optimal in any context.

On the other hand, very little attention has been paid by the ILP community to the issue of minimising the test costs of the learned models. An adaptation of this system to ILP named RL-ICET (relational learning with ICET) was defined in [Turney, 1995b]. The algorithm consists of three steps: First, each instance of the problem to be learned is automatically translated into a feature vector. A test cost is assigned to every feature of the vector depending on the complexity of the code required to translate that feature. Secondly, ICET generates a decision tree from the feature vectors and the corresponding cost. Finally, the decision tree is converted into Prolog code. Since ICET produces low cost decision trees, in this case it produces low size complexity programs. RL-ICET was successfully applied in the machine learning contest East/West Challenge [Michie *et al.*, 1994]. A similar algorithm (without costs) is employed by the ILP system Linus [Lavrac *et al.*, 1991; Lavrac and Dzeroski, 1994] (see Section 3).

Chapter 6

Combination of Models

In this chapter we introduce a topic in machine learning that is getting much attention from the machine learning community during the last years. In particular, we discuss about some techniques that combine models in order to improve the accuracy of the resulting model. These techniques are called *ensemble methods*. In this chapter, we will restrict our discussion to classification problems.

6.1 Introduction

With the goal of improving model accuracy, there has been an increasing interest in defining methods that combine hypotheses. These methods construct a set of hypotheses (ensemble), and then combine the components of the ensemble in some way (typically by a weighted or unweighted voting) in order to classify examples. The accuracy obtained will be often better than that of the individual components of the ensemble. This technique is known as *Ensemble Methods* [Dietterich, 2000a].

The quality of the generated ensemble highly depends on the accuracy and diversity of its individual components [Hansen and Salamon, 1990]. For instance, consider an example [Dietterich, 2000a] of an ensemble of three classifiers: $\{h_1, h_2, h_3\}$, and consider a new example x. If the three classifiers are similar, then when $h_1(x)$ is wrong, $h_2(x)$ and $h_3(x)$ will be likely be wrong. However, if the classifiers are diverse, the errors they commit will be uncorrelated, then when $h_1(x)$ is wrong, $h_2(x)$ and $h_3(x)$ may be correct, so that a majority vote will correctly classify x.

The combined hypothesis is richer in expressiveness than any of its components. A learning algorithm is employed to find different hypotheses in the hypothesis space or language $\mathcal{H}$. By constructing an ensemble out of all these classifiers, the algorithm can "average" their votes, and the ensemble hypothesis can be outside of $\mathcal{H}$. Figure 6.1 depicts this situation. The outer curve denotes the hypothesis space $\mathcal{H}$. The

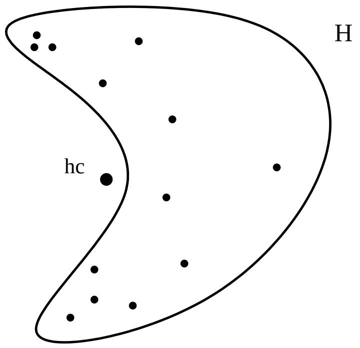


Figure 6.1: Representation of an ensemble of hypotheses

points represents the components of the ensemble, and h_c is the ensemble classifier.

Other advantage of ensemble methods is that the combination of different and diverse models usually avoids the problem of overfitting the training data.

Hence, there are some important issues that affect on the final quality of an ensemble: their diversity, their construction and their combination. In the following, we address these issues. Let us begin discussing measures for evaluating their diversity.

6.1.1 Hypothesis Similarity Metrics

As we have discussed, an important factor of the quality of an ensemble is the diversity of its components. Consequently, several measures of hypothesis similarity (or diversity or fidelity) have been considered in the literature with the aim of obtaining an ensemble with high diversity [Kuncheva and Whitaker, 2002]. In this section, we summarise some measures defined to establish the diversity between two hypotheses.

Given two classifiers h_a and h_b, and an unlabeled dataset with n examples with C classes, we can construct a $C \times C$ contingency or confusion matrix $M_{i,j}$ that contains the number of examples e such that $h_a(e) = i$ and $h_b(e) = j$. Using this matrix we can define the following similarity metrics:

- θ **measure**: It is just based on the idea of reckoning the probability that both classifiers agree:

$$\theta = \sum_{i=1}^{C} \frac{M_{i,i}}{n}$$

  Its value is between 0 and 1. An inverse measure, known as discrepancy is also considered by [Kuncheva and Whitaker, 2002].

- κ **measure**: The previous metric has the problem that when one class is much more common than the others or there are only two classes, this measure is highly affected by the fact that some predictions may match just by chance. Following [Margineantu and Dietterich, 1997], we define the Kappa measure, originally introduced as the Kappa statistic (κ) [Cohen, 1960]. This is just a proper normalisation based on the probability that two classifiers agree by chance:

$$\theta_2 = \sum_{i=1}^{C} \left(\sum_{j=1}^{C} \frac{M_{i,j}}{n} \cdot \sum_{j=1}^{C} \frac{M_{j,i}}{n} \right)$$

  As a result, the Kappa statistic is defined as:

$$\kappa = \frac{\theta - \theta_2}{1 - \theta_2}$$

  Its value is usually between 0 and 1, although a value lower than 0 is possible, meaning that the two classifiers agree less than two random classifiers.

- Q **measure**: The Q measure is defined as follows [Kuncheva and Whitaker, 2002]:

$$Q = \frac{\prod_{i=1}^{C} M_{i,i} - \prod_{i=1,j=1,i\neq j}^{C} M_{i,j}}{\prod_{i=1}^{C} M_{i,i} + \prod_{i=1,j=1,i\neq j}^{C} M_{i,j}}$$

  This value varies between -1 and 1. Note that this measure may have problems if some components of M are 0. Thus, it is convenient to apply smoothing in M to compute the measure.

Obviously, any of the previous metrics would give a better estimate of similarity the greater the reference dataset is.

6.2 Methods for Constructing Ensembles

Many methods have been proposed to construct a set of classifiers from a single evidence. These techniques have been applied in many different learning algorithms. Dietterich [Dietterich, 2000a] distinguishes different kinds of ensemble construction methods, being the following ones the most frequently used:

1. **Manipulating the training examples**: These methods consist in running several times the same learning algorithm, each time with a different subset of the training examples, thus generating, a different classifier for any set. The

relevant issue in these methods is to define a good mechanism to generate subsets from the set of training examples. For instance, *Bagging* [Breiman, 1996; Quinlan, 1996a], *Boosting* [Freund and Schapire, 1996; Quinlan, 1996a] and *Cross-validated committees* [Parmanto *et al.*, 1996] are ensemble methods of this family.

2. **Manipulating the input features**: A general technique to obtain the components of an ensemble is to alter the set of input attributes to the learning algorithm. The forest technique [Ho, 1998] belongs to this category.

3. **Manipulating the output features**: Other method consists in manipulating the classes of the examples in the training dataset. [Dietterich and Bakiri, 1995] defined a technique that generates datasets by randomly partitioning the K classes of the original dataset into two subsets A and B. Then, it relabels the classes assigning class 0 to subset A, and class 1 to subset B. With this modified dataset, it learns a classifier. By repeating this process several times, an ensemble of classifiers is obtained. To employ the ensemble to classify new examples a special voting is performed in order to select a class among the original k classes.

4. **Random methods**: The last group of ensemble methods is formed by the techniques that inject randomness into the learning process to obtain different classifiers from the same training dataset. This method has been employed in the backpropagation algorithm of neural networks by setting the initial weights randomly [Parmanto *et al.*, 1996]. It is also feasible to apply randomness in decision tree learning; the *randomisation* method [Dietterich, 2000b] introduces randomness in the internal split selection during the decision tree construction.

Due to the importance and its wide application, in the following we introduce the most well-known ensemble methods.

Bagging

Bagging [Breiman, 1996; Quinlan, 1996a] is a simple and powerful method for generating ensembles. *Bagging* is derived from the technique *bootstrap aggregation*. This method constructs subsets by generating a sample of m training examples, selected randomly (and with replacement) from the original training set of m components. The new subsets of training examples are called *bootstrap replicates*.

Boosting

The mechanism that employs *Boosting* [Freund and Schapire, 1996; Quinlan, 1996a] to generate the components of the ensemble is based on the assignment of a weight to each example of the training dataset. In each iteration, the method learns a hypothesis that minimises the weighted error. The errors obtained are used to update the weights of the examples, in particular to increase the weight of the examples which have been misclassified, and to decrease the weight of the examples classified correctly. With the updated weights a new hypothesis is constructed, giving more relevance to the examples for which the previous hypothesis behaved badly. This process is applied as many times as the required number of components of the ensemble. *AdaBoost* [Freund and Schapire, 1996, 1997] is a popular implementation of this ensemble method.

Randomisation

A recent ensemble method defined in [Dietterich, 2000b] is known as *Randomisation*. This method is based on introducing randomness in the internal decisions of the learning process. In particular, it has been defined for decision tree learning. The idea is selecting randomly among the top 20 best splits. In this way, the method generates a different decision tree for each iteration.

6.2.1 Ensemble Methods Comparison

Dietterich has compared experimentally these three ensemble methods in [Dietterich, 2000b]. The conclusions are that in problems without noise *Boosting* gets the best results, while the results of *Bagging* and *Randomisation* are quite similar. With respect to the noisy datasets, *Bagging* is the best method, followed by *Randomisation*, and, finally, *Boosting*.

Other work that compares ensemble methods is [Bauer and Kohavi, 1999]. This work is concentrated on comparing empirically *Boosting*, *Bagging* and various of their variants. The authors employ two families of learning methods: decision trees (concretely, C4.5) and Naive-Bayes [Duda and Hart, 1974]. The best ensemble algorithm was *AdaBoost* [Freund and Schapire, 1996, 1997]. This algorithm reduced 26% of the average error for C4.5, and 24% of the average error for Naive-Bayes. They also observed that *Boosting* was generally better than *Bagging*, but not uniformly better.

Finally, [Maclin and Opitz, 1997] obtains similar results comparing *Boosting* and *Bagging* with decision trees and neural networks. They stated that *Boosting* seems to obtain better accuracy in general; however, *Bagging* methods are more robust with noisy problems.

6.3 Vector Fusion Strategies

We have discussed about how to generate the components of the ensemble. Once the ensemble has been constructed, this will be employed for classifying new examples.

However, there are many methods to compute the final class of an example given a set of classifiers. In this section, we address this issue.

If we consider all the estimates equally reliable (or unreliable), several fusion strategies have been employed. Let us define a *probability vector* $v_{k,j}(x)$ as the vector of training cases that fall in a rule k for each class j. Following [Kuncheva, 2002], there are several fusion strategies that convert m probability vectors into one combined vector α_j:

- **sum**: $\alpha_j = \sum_{k=1}^{m} v_{k,j}$
- **arithmetic mean**: $\alpha_j = \sum_{k=1}^{m} \frac{v_{k,j}}{m}$
- **product:** $\alpha_j = \prod_{k=1}^{m} v_{k,j}$
- **geometric mean:** $\alpha_j = \sqrt[m]{\prod_{k=1}^{m} v_{k,j}}$
- **maximum:** $\alpha_j = \mathtt{max}_k(v_{k,j})$
- **minimum:** $\alpha_j = \mathtt{min}_k(v_{k,j})$
- **median:** $\alpha_j = \mathtt{median}_k(v_{k,j})$

There have been some studies to examine which strategy is better. In particular, [Kuncheva, 2002] concludes that, for two-class problems and under certain assumptions, minimum and maximum are the best strategies, followed by average (arithmetic mean).

Let us illustrate the previous measures with an example.

Example 8 *Consider four rules of a class-problem with the following values:*

$$
\begin{aligned}
v_{1,j}(x) &= \{40, 10, 30\}\\
v_{2,j}(x) &= \{7, 2, 10\}\\
v_{1,j}(x) &= \{0, 10, 1\}\\
v_{1,j}(x) &= \{5, 6, 3\}
\end{aligned}
$$

This would give the following fusion vectors:

$$
\begin{array}{lll}
sum: & \{52, 28, 44\} & \rightarrow a\\
arithmean: & \{13, 7, 11\} & \rightarrow a\\
product: & \{0, 1200, 900\} & \rightarrow b\\
geomean: & \{0, 5.89, 5.48\} & \rightarrow b\\
maximum: & \{40, 10, 30\} & \rightarrow a\\
minimum: & \{0, 2, 1\} & \rightarrow b\\
median: & \{6, 8, 6.5\} & \rightarrow b
\end{array}
$$

On the right we see which class would be predicted if we compute the maximum value of the resulting fusion vector (i.e "majority vote").

6.4 Drawbacks of Ensemble Methods

As we have seen, ensemble methods represent a powerful and a relatively simple technique to obtain an accurate classifier from a combination of a set of classifiers. However, this technique also has some problems.

The ensemble method classifies new instances by combining the results of its components. However, this combination results in a complete loss of comprehensibility. A combined hypothesis is usually a voting of many hypotheses and it is usually treated as a black box, giving no insight at all.

Another problem is the high amount of resources required to construct, store and use the ensemble. In [Margineantu and Dietterich, 1997], one of these problems is highlighted, "the large amount of memory required to store these hypotheses can make ensemble methods hard to deploy in applications". They show that only with 20% to 40% of the whole set of different hypotheses, a combined hypothesis can have the same accuracy. In [Quinlan, 1998b], a method called miniboosting reduces the ensemble to just three hypotheses, with 40% less of the improvement that would be obtained by a 10-trial AdaBoost.

Ensemble methods also require to test each of their components in order to classify just one instance. This results in a significant decrease of classification throughput (the speed in which an unknown instance can be classified). This is very relevant in real-time applications.

Finally, another problem related to ensemble methods is the necessity of testing the majority or all of the attributes of an example in order to predict its class. In a single classifier it is possible to predict the class of an example testing only a subset of the attributes. However, in an ensemble, all of its classifiers must be executed, then the number of attributes to be tested will be increased with respect to the size of the ensemble. This drawback is especially important when the attributes have costs associated to their test. For instance, it is more expensive to perform a blood analysis to a patient than to ask his age. Ensemble methods will perform both things.

6.5 Ensemble Methods and Relational Learning

In propositional learning, ensemble methods have been a very popular technique for improving the accuracy of classification learners. However, in relational learning, not much attention has been paid to ensemble methods, mainly due to the loss of comprehensibility associated to these methods, and because usually ILP methods are too slow to repeat many iterations of the learning algorithm.

The first approach to ensemble methods employing relational learning was presented by Quinlan in [Quinlan, 1996b, 2001]. In this work, *Boosting* is applied to

FFOIL (see Section 4.2.1). This approach reduces the error rate by an average of 13% with respect to basic FFOIL. Hence, he concludes that relational learners can also take profit of the advantages of ensemble methods to improve the accuracy of models. However, the produced models sacrifice the comprehensibility of relational learning.

This hindrance is partially addressed by the CCRB (Constrained Confidence-Rated Boosting) algorithm [Hoche and Wrobel, 2001]. In this case, the authors apply *Boosting* to a fast weak ILP learner. They compare this algorithm with other ILP systems, and the results are quite identical in accuracy, although the use of the weak ILP learner allows to reduce the consumption of resources. However, although the comprehensibility of the models produced by this algorithm is better than those generated by the Quinlan's approach, they are more obfuscated than ILP rules.

6.6 Cost-sensitive Ensemble Methods

Cost-sensitive learning has also been employed for the construction of ensemble of classifiers. An adaptation of *Adaboost* for a cost-sensitive context is known as *Adacost* [Fan *et al.*, 1999]. In this version, the misclassification costs are employed to update the weights of the examples during the execution of the algorithm.

[Margineantu, 1999] introduces two methods that allow *Bagging* and *Boosting* to be cost-sensitive. The technique for *Bagging* is based on the assignment of a weight to every example depending on the cost matrix by using Powell's method [Press *et al.*, 1992], then *Bagging* votes the decision trees generated by using bootstrap samples of these weighted examples. The method employed for making *Boosting* cost-sensitive is quite similar to *Adacost* in the sense that the weight of the examples depends on the mis-classification cost.

Regarding test costs, ensemble methods are the worst situation possible; since many (and different) models are needed, it is very likely that all possible tests should be performed. Therefore, the use of ensemble methods when there are costs associated to the tests is inadvisable. Some other methods such as neural networks, Bayesian methods and support vector machines have the same problem. Decision tree learning is, in this regard, the best approach due to the fact that often the predictions of an example can be made without testing all of its attributes. Moreover (as we have seen in Section 5.2.4), there are some approaches that construct decision trees that minimise test costs.

Part II

Induction of Functional Logic Programs (IFLP)

Chapter 7

A Framework for IFLP

In this chapter we present a framework for the learning of functional logic programs. We describe the operators and the algorithm of the approach. We also define an incremental extension of the framework.

7.1 Introduction

The use of logic programming for learning is mainly based on the idea that logic programs are a single and suitable representation for examples, background knowledge and hypotheses. As we have shown, ILP has provided an outstanding advance in machine learning by increasing the applicability of learning systems to theories with more expressive power than propositional frameworks. However, logic languages like Prolog (the most representative language of this paradigm) lack some programming facilities such as evaluable and nested functions, types, higher order programming and lazy evaluation. Although these features are well supported by functional languages, these do not enjoy the computing power provided by logical variables and unification. Hence, the interest in the integration of both families of languages has grown over the last few years.

As we discussed in Section 4.2.1, integrated languages fully exploit the facilities of logic programming in a general sense: functions, predicates and equality. One relevant approach to integration is functional logic programming, where the programs are logic programs that are augmented with Horn equational theories. It is widely accepted that narrowing is the key to describing the operational semantics of functional logic languages.

In this chapter, we present the Inductive Functional Logic Programming paradigm (IFLP). The theory of IFLP is based on generalisation and on the inversion of the transitivity property of equality, leading to two operators: consistent restricted generalisation and inverse narrowing. These are combined in an evolutionary-like algorithm which works with a population of equations and programs. This framework has been presented in [Hernández and Ramírez, 1998, 1999; Ferri *et al.*, 2000a, 2003].

In order to improve performance we have also extended the main algorithm to handle the evidence in an incremental way. Incrementality is a powerful and useful technique in machine learning. A learning system is incremental if the examples are supplied either one or a few at a time and after each example the system induces, maintains or revises a hypothesis. This operational mode is opposite to that of non-incremental systems (known as batch systems) for which the entire evidence is given initially and does not change afterwards. A description of this extension has been published in [Ferri *et al.*, 2001b].

The chapter is organised as follows. In Section 7.2, we define the IFLP paradigm. Section 7.3 defines the IFLP operators, especially the inverse narrowing operator. Section 7.4 presents an algorithm for the inductive inference of functional logic programs based on the inverse narrowing approach. Section 7.5 introduces an incremental version of the algorithm. Finally, Section 7.6 concludes the chapter.

7.2 IFLP Paradigm

IFLP can be defined as the functional (or equational) extension of ILP[1]. The goal is the inference of a theory (an unconditional functional logic program P) from evidence (a set of positive and optionally negative ground equations E) using a background knowledge theory (a functional logic program B) such that the following conditions hold:

1. $B \not\models E^+$ (*prior necessity*)
2. $\forall\, e^- \in E^- : B \not\models e^-$ (*prior satisfiability*)
3. $B \cup P \models E^+$ (*posterior sufficiency*)
4. $\forall\, e^- \in E^- : B \cup P \not\models e^-$ (*posterior satisfiability*)

Properties 3 and 4 are also referred as *completeness* and *negative consistency*, respectively. We say that a theory P *covers* an example e iff $P \models e$.

Since the aim now is to induce functions, an additional condition is needed: positive consistency. Given a set of equations E^+ and a functional logic program P, we say that P is *positive consistent* with respect to E^+ if for all $l = r \in E^+$, $\lfloor l \rfloor \cap \mathcal{T}(\mathcal{C}) = \{r\}$ or $\lfloor l \rfloor \cap \mathcal{T}(\mathcal{C}) = \varnothing$ (see Section 4.1 for the notation).

[1] It is obvious that any problem expressed in the ILP framework can also be expressed in the IFLP framework, because all the positive facts e_i^+ of an ILP problem can be converted into equations of the form $e_i^+ = true$ and all the negative facts e_j^- can be expressed as $e_j^- = false$.

We will consider evidence E composed of (positive E^+ and negative E^-) equations[2] whose rhs are normalised w.r.t. the background theory B and the theory P which is meant to be discovered (target hypothesis), with $B \cup P$ being canonical. B must always hold the prior necessity and prior satisfiability conditions w.r.t. E. We can do some preprocessing to E, which includes well known equation simplifications: (i) any equation of the form $f(x_1, x_2, \ldots, x_n) = f(y_1, y_2, \ldots, y_n)$ is replaced by the n equations $x_1 = y_1$, $x_2 = y_2$, $\ldots$, $x_n = y_n$, and (ii) the elimination of all redundant equations (after standard renaming).

In logic programming, the induction can be made top-down (starting from the most general program and refining it by specialisation) or bottom-up (starting from positive data as a program and generalising it). In the case of functional logic programs, we cannot follow a top-down direction because the examples are equations, and the most general program $X = Y$ would make the program non-terminating. On the other hand, the most specific generalisation in this case is the positive evidence itself.

7.3 IFLP Operators

In this section, we present the main operators for the IFLP framework: consistent restricted generalisation and inverse narrowing. Both operators generate new equations which are used by the IFLP algorithm to construct programs.

7.3.1 Consistent Restricted Generalisation

Our first operator is inspired by the least general generalisation approach of Plotkin, which is also used in ILP (see Section 3.2). Since we have to ensure consistency, we begin generating all possible restricted generalisations from each positive example which are consistent with both positive and negative examples. More formally,

Definition 3 (Restricted Generalisation, RG.) *An equation $t = s$ is a restricted generalisation (RG) of an equation $r = m$ if $\exists \sigma : \sigma(t) = r \land \sigma(s) = m$, $\forall x (x \in Var(s) \Rightarrow x \in Var(t))$ and the term t is not a variable.*

In other words, RG does not introduce extra variables on the rhs of the equations. RG prevents meaningless generalisations from being considered when we search the target program for the given examples.

Definition 4 (Consistent Restricted Generalisation, CRG.) *An equation e' is a consistent restricted generalisation (CRG) of e w.r.t. an evidence E and an existing*

[2]If only Boolean functions were to be induced, E^- could be expressed inside E^+ since every negative evidence can be expressed as a positive one.

theory $T = B \cup P$ if and only if e' is a RG of e and $T \cup \{e'\}$ is positive and negative consistent w.r.t. E.

The set of CRGs of an equation is generated from a tree called the occurrence tree of the equation. In the tree, the different occurrences of the equation that denote the same subterm are organised as equivalence classes. Hence, the branches of the tree are labelled by the occurrences and the nodes are labelled by their corresponding equivalence classes. Only equivalence classes which are obtained from settled non-variable occurrences are considered.

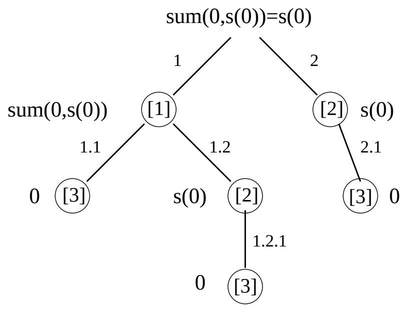


Figure 7.1: Occurrence tree of $sum(0, s(0)) = s(0)$.

For instance, in Figure 7.1, we can see the occurrence tree of the equation e : $(sum(0, s(0)) = s(0))$. In this example, we have three different equivalence classes: $[1] = \{sum(0, s(0))\}$, $[2] = \{s(0)\}$, and $[3] = \{0\}$.

In addition, we associate a buffer to each equivalence class. This buffer, which is initially empty, is filled with the variables that we assign to the occurrences of this class during the process.

Briefly, the CRGs of an equation e are obtained as follows. The occurrence tree is traversed (depth-first search, left to right) building a new tree whose leaves are the RGs of e. In each node, an occurrence of e which was not previously considered is handled. The children are new equations generated by replacing the subterm of e at the selected occurrence by a new variable and by the variables in the corresponding buffer. We also consider an additional children where the selected ocurrence is not replaced. Finally, the right branch analyses the rest of occurrences of e.

Figure 7.2 shows the "RG tree" which is produced by this process from the equation $sum(0, 0) = 0$. The content of the buffers is included in each branch of the tree. The equations marked with a dot are the valid RGs (without extra variables on the right hand side).

To obtain the CRGs, we prove the positive and negative consistency of each restricted generalisation g in the RG tree as follows:

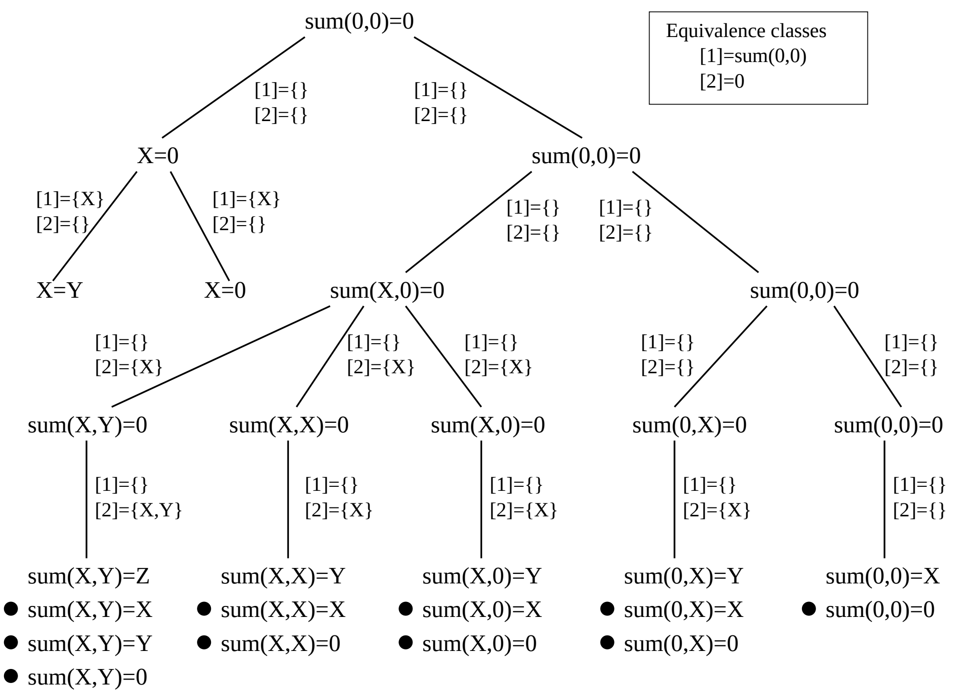


Figure 7.2: RG tree of the equation $sum(0,0) = 0$.

- for each $l = r \in E^+$, $\mathtt{cnf}_{\{g\}}(l) = \{r\}$ or $\mathtt{cnf}_{\{g\}}(l) = \varnothing$
- for each $l = r \in E^-$, $r \notin \mathtt{cnf}_{\{g\}}(l)$

Consequently, we reject those equations that are positive or negative inconsistent.

Positive Examples	Negative Examples
sum(0,0)=0	sum(s(0),0)=0
sum(s(0),s(0))=s(s(0))	sum(0,0)=s(0)
sum(0,s(0))=s(0)	sum(s(0),s(0))=s(0)
sum(s(s(0)),0)=s(s(0))	sum(s(0),0)=s(s(0))

Table 7.1: Example clauses.

Table 7.2 illustrates how all the restricted generalisations are filtered, until the correct CRGs are obtained. The example which is computed is $sum(0,0) = 0$, and the positive and negative consistency is proved w.r.t. the examples in Table 7.1. The first column of Table 7.2 contains the set of the equations that satisfy the RG specification; and the second and third columns include the equations that hold the positive and negative

Restrictive Generalization (RG)	Negative Consistency	Positive Consistency
sum(0,0) = 0	sum(0,0)=0	sum(0,0)=0
sum(0,X) = 0	sum(0,X) = 0	sum(0,X) = X
sum(0,X) = X	sum(0,X) = X	sum(X,0) = X
sum(X,0) = 0	sum(X,0) = X	
sum(X,0) = X	sum(X,X) = 0	
sum(X,Y) = 0		
sum(X,Y) = X		
sum(X,Y) = Y		
sum(X,X) = 0		
sum(X,X) = X		

Table 7.2: CRGs extracted from the equation sum(0,0)=0 w.r.t. the examples in Table 7.1.

consistency filters, respectively. The equations of the third column are the CRGs.

7.3.2 Inverse Narrowing

Since narrowing is a sound and complete method for $\mathcal{E}$-unification, in order to define an inverse narrowing operator, we start analysing the properties of the equality relation, in particular, the transitive property. In what follows, for the sake of legibility, we consider x, y and z to be subterms at the occurrence Λ. The following rationale is still valid for any other occurrence in a term.

The transitive property is expressed as[3]:

$$x \to y \wedge y \to z \Rightarrow x \to z \tag{7.1}$$

To have a constructive inversion of the transitivity of equality, the behaviour of the operator should be as follows:

$$x \to z \Rightarrow x \to y \wedge y \to z \tag{7.2}$$

where $x \to y$ and $y \to z$ are the result of this constructive inverse process. Notice that the term y in the above formula (7.2) is new (i.e., it can be considered as an auxiliary function symbol if it belongs to the signature of the problem or as an invented function symbol if it does not).

The wide generality of the previous approach makes it intractable from a computational point of view. Let us note that, for instance, there is an infinite number of

[3] $\to$ is used for equality.

terms which can be used as the y term in formula 7.2 if the signature has, at least, one function symbol with arity ≥ 1. To overcome this problem, we have studied an approximation that we will show to be appropriate for practical purposes. This process can be seen as an operator that inverses the narrowing mechanism, so we call it *inverse narrowing.*

Consider formula 7.2 again. One possibility is to restrict y to be a term which is related in some way to the positive evidence. Therefore, the idea is to supply this y, giving a different view:

$$x \to z \wedge y \to z \Rightarrow x \to y \wedge y \to z \tag{7.3}$$

where $x \to y$ in formula 7.3 is the equation to be computed from $x \to z$ and $y \to z$. However, formula 7.3 is not a real inversion of transitivity because it begins from two equations (one of the premises is the result of formula 7.1 and it generates its other premise). In accordance with formula 7.3, this operator will allow us to generate the equation $x \to y$ from the equations $x \to z$ and $y \to z$.

Definition 5 (Reverse Narrowing.) *Given a functional logic program P, we say that a term t reversely narrows into a term t', and we write $t \overset{u,l=r,\theta}{\hookleftarrow}_P t'$, iff $u \in O(t)$, $l = r$ is a new variant of a clause from P, $\theta = mgu(t_{|u}, r)$ and $t' = \theta(t[l]_u)$. The relation $\hookleftarrow_P$ is called the reverse narrowing relation.*

Example 9 *Suppose we are inducing a program P from the following examples which define the addition operator for the natural numbers:*

		(E_1^-)	$s(0) + 0 = 0$
(E_1^+)	$0 + 0 = 0$	(E_2^-)	$0 + 0 = s(0)$
(E_2^+)	$s(0) + s(0) = s(s(0))$	(E_3^-)	$s(0) + s(0) = s(0)$
(E_3^+)	$0 + s(0) = s(0)$	(E_4^-)	$s(0) + 0 = s(s(0))$
(E_4^+)	$s(s(0) + s(0)) = s(s(s(0)))$	(E_5^-)	$s(0 + 0) = s(s(0))$
(E_5^+)	$s(s(s(0)) + s(0)) = s(s(s(s(0))))$	(E_6^-)	$s(s(0) + s(0)) = 0$
		(E_7^-)	$s(0) = 0$

Suppose we select the clause $X' + 0 = X'$ as a good CRG to be part of P and we arbitrarily select the rhs of another CRG clause $X + s(0) = s(X)$, i.e., $s(X)$. The first clause can be used inversely in $s(X)$ in different positions. In this case, there are two different possible applications corresponding to a variable and a non-variable occurrence:

(t_1) $s(X + 0)$
(t_2) $s(X) + 0$

That is to say, t_1 and t_2 can be narrowed to $s(X)$ using a clause of P.

Let us now present a more complex operator: inverse narrowing.

Inverse narrowing uses the terms computed by reverse narrowing to generate new equations. For instance, in Example 9, from the rhs of the rule $X + s(0) = s(X)$ we have obtained the terms $t_1 : s(X + 0)$ and $t_2 : s(X) + 0$ by reverse narrowing ($s(X) \hookleftarrow_P t_1$ and $s(X) \hookleftarrow_P t_2$). Then, the term $s(X)$ in $X + s(0) = s(X)$ can be replaced by t_1 and t_2, obtaining the new equations $X + s(0) = s(X + 0)$ and $X + s(0) = s(X) + 0$.

Example 10 *Suppose that the member program is to be induced. This program determines whether an element is a member of a list. Consider that the lists are represented via the constructor λ, which represents the empty list, and the constructor $\cdot(List, Element)$.*

If the consistent equations, $member(\cdot(\cdot(X,Y),Z),Y) = true$ and $member(\cdot(X,Y),Y) = true$, are chosen for applying reverse narrowing, we will obtain $member(\cdot(\cdot(X,Y),Z),Y) = member(\cdot(A,B),B)$ because $true \hookleftarrow member(\cdot(A,B),B)$.

Due to the fact that reverse narrowing applies the selected equation from right to left, new variables could be introduced in the resulting term (as we can see in the above example 10). Therefore, the inverse narrowing mechanism must detect the possible appearance of such extra variables in the new equations. The direct rejection of these equations could produce the loss of a good way to progress towards the solution program. Hence, we have designed a method that permits the removal of the extra variables by binding extra variables to variables on the lhs. This variable binding is done in such a way that every possible combination is performed, being careful that two different extra variables have not been bound to the same lhs variable. That is, this is equivalent to the permutations without repetitions of the lhs variables over the extra variables on the rhs. Therefore, if we have n variables on the lhs, and m extra variables, $\frac{n!}{(n-m)!}$ different equations will be obtained. If the equation has more extra variables than variables on the lhs, it is rejected. The following example illustrates the process.

Example 11 *Continuing with Example 10, the equation*

$$member(\cdot(\cdot(X,Y),Z),Y) = member(\cdot(A,B),B)$$

contains extra variables (A and B) which can be replaced by the variables on the lhs, producing the following equations:

$$\begin{aligned}
member(\cdot(\cdot(X,Y),Z),Y) &= member(\cdot(X,Y),Y)\\
member(\cdot(\cdot(X,Y),Z),Y) &= member(\cdot(X,Z),Z)\\
member(\cdot(\cdot(X,Y),Z),Y) &= member(\cdot(Y,X),X)\\
member(\cdot(\cdot(X,Y),Z),Y) &= member(\cdot(Y,Z),Z)\\
member(\cdot(\cdot(X,Y),Z),Y) &= member(\cdot(Z,X),X)\\
member(\cdot(\cdot(X,Y),Z),Y) &= member(\cdot(Z,Y),Y)
\end{aligned}$$

Note that although some of these equations can contain terms that do not satisfy the type profile of the function (for instance, the rhs of the third equation in the above example), they are removed later because the consistency property does not hold.

Finally, since the narrowing mechanism performs a unification followed by a replacement, the inversion of this deductive operator must be based on a reverse narrowing step followed by a CRG operation.

Definition 6 (Inverse Narrowing.) *Given two equations, $s = t$ known as sender rule, and $r = v$ known as receiver rule, all possible reverse narrowing steps that can be applied in v w.r.t. the sender rule are made. Let n be the terms computed by reverse narrowing. Then, for each $v_i, 1 \leq i \leq n$, ($v \overset{i,s=t,\theta_i}{\hookleftarrow} v_i$) a rule $r = v_i$ is built. Next, extra variables are bound and the CRGs of each resulting rule are calculated. Finally, the output of the inverse narrowing operator is a set of pairs of equations formed by the sender rule and a CRG rule.*

Example 12 *Consider Examples 10 and 11 again. Applying the CRG operator to the equation $member(\cdot(\cdot(X,Y),Z),Y) = member(\cdot(X,Y),Y)$, we will get the equation $member(\cdot(A,B),C) = member(A,C)$, that together with the equation $member(\cdot(X,Y),Y) = true$ constitute the solution program.*

7.4 General Algorithm and Optimality Criteria

In this section, we discuss the skeleton of the algorithm for the inductive inference of functional logic programs. Our learning task consists of an inductive search of hypothetical equations as well as a selection of programs which are constructed from these equations until one of the programs is evaluated as a good solution.

As we have already mentioned, we start the inductive process from positive and negative evidence E^+ and E^-. Additionally, a background theory B can be used to induce the target program P. In the following, we will denote BF (Basic Functions) to be the subset of functions from B that are determined by the user and which can be used in the equational clauses of P.

The algorithm is driven by a flexible optimality criterion. The possibility of changing this criterion (which guides the search for the solution) forces us to use a flexible algorithm structure, in particular, a skeleton which resembles evolutionary programming techniques [Fostel, 1995]. Genetic or evolutionary techniques have also been used by the ILP community [Ichise, 1998; Varsek, 1993]. The similarities between these and our algorithm are:

- Evolving Population: there is an evolving population of individuals, from which the best solution is extracted.

- Crossing Operators: several operators are provided to combine different individuals.

- Fitness Criterion: an optimality criterion is used to determine the goodness of an individual.

- Selection: individuals for the crossing operators are selected according to the fitness criterion.

However, some important differences distinguish our algorithm from preceding ones:

- Two levels of populations: equations and programs. In evolutionary terminology, this could be understood as individuals and communities of individuals. Communities are crossed by choosing the best of their individuals. However, the solution of the algorithm is a community, not an individual.

- Two levels of operators: CRG for individuals (mutation) and inverse narrowing for communities (crossing).

- Memory: communities that have been crossed are not crossed again. As long as communities do not get too numerous, this allows for a much more efficient algorithm with more varied genotypes than if memory were not considered.

- Determinism: presently, no random elements are included. In other words, if the same problem is supplied to the algorithm twice, the result is always the same.

- No twins: although one individual can appear in more than one community, communities are different from each other and twins are not allowed in the same community. In other words, redundant clauses are removed.

- No death: for completeness reasons, there is no population limit. Consequently, death has not been considered for the moment.

- External restrictions: not every combination yields a valid individual. In our case, consistency, confluence and termination are external restrictions that reject many possible individuals and/or communities.

The IFLP algorithm is a particularisation of the preceding features. Note that the optimality criterion is independent from the structure of an evolutionary algorithm and independent from the operators that may be used (to an extent). This is precisely what is intended. Next, the optimality criterion is discussed.

7.4.1 Hypothesis Selection

As in ILP, we have to select "the optimal program" from all the many possible *valid* programs that ensure completeness and consistency. The classical ML problem is that there is no such thing as "the right hypothesis", so an optimality criterion must be arbitrarily selected depending on the application or purpose of the induction: prediction, scientific discovery, program synthesis, function invention, program transformation, abduction or explanation-based learning (EBL), etc. Moreover, some of them drastically diverge for different kinds of samples: (perfect / imperfect), (complete / incomplete) and (positive evidence only / positive and negative evidence).

Despite this undeniable fact, the *Minimum Description Length (MDL)* principle (see Section 5.1.1) is the most popular selection criterion in ILP. This is because it is supported by the classical view of unsupervised learning as compression, supported by the effectiveness of its use in many applications of ML, and is also supported by its recent formal justification [Li and Vitányi, 1997] in relation to Bayesian learning. The MDL principle has been successfully applied mainly where the source has a statistical character and may contain errors. However, in other applications where no errors are expected from the source [Li and Vitányi, 1997], such as program synthesis from examples or in *incremental* learning, the MDL principle sometimes is much too conservative. In this section, we present the simplest criterion we have essayed which is a concretisation of the MDL principle. In particular, we will compute the length of an equation e as

$$length(e) = 1 + n_v/2 + n_c + n_f$$

with n_v, n_c and n_f being, respectively, the number of variables, constants and functors of only the rhs of e, as it is desirable to obtain short equations with decreasing character[4]. Note that we promote variables over constants or functors. Finally, we define the information factor of a set of equations P as

$$Inf(P) = -\sum_{e \in P} log_2 length(e)$$

The *covering factor* w.r.t. the positive evidence, $CovF^+$, is defined as

$$CovF^+(P) = card(e \in E^+ : P \models e)/card(E^+)$$

i.e., the proportion of positive cases covered.

Specifically, our optimality measure is constructed as[5]:

[4]Theoretically, the length should be computed using a generating grammar for all the terms in the Herbrand Universe as in [Conklin and Witten, 1994]. In our case, we give an approximation that works well in practice and has the advantage of being independent of the total number of constants, variables and functors in the program or the Herbrand Universe.

[5]For the time being we have just considered $\alpha = 1$, $\beta = 1$, but, in the future, these values could be parametrised for different kinds of problems.

$$Opt(P) = \alpha \times Inf(P) + \beta \times CovF^{+}(P)$$

Once the hypothesis selection criterion is settled, the algorithm drives the generation of new equations and programs in a proper way, using the optimality criterion as a search heuristic. Note that efficiency is implicitly taken into account due to the character of the search. The stop criterion used can also be independently selected. For instance, in the case of considering perfect data (no transmission errors or noisy data), the stop criterion consists only of the completeness condition $CovF^{+} = 1$ and the syntactical length criterion. This makes our approach very generic and easily adaptable to quite different applications.

7.4.2 The IFLP Algorithm

The basic IFLP algorithm learns programs by generating two sets: a set of individuals (a set of equations denoted by EH, Equation Hypotheses) where the equations are initially generated by means of the CRG operator, and a set of communities (a set of programs denoted by PH, Program Hypotheses) which are composed exclusively from equations of EH. At each step of the algorithm, new equations and programs are generated by inverse narrowing. Thus, the kernel of the algorithm is constituted by an auxiliary procedure: *InverseNarrowing*.

The Procedure *InverseNarrowing(input:*P_1, P_2, BF*; output:*EH, PH*)* returns a set of equations (EH) and a set of programs (PH) which are obtained by applying inverse narrowing between the equations of the two input programs P_1 and P_2.

The first step of the learning algorithm generates the initial EH set with all the CRGs from E^{+} (procedure *GenerateCRG*). Next, PH is initialised to the set of all possible programs containing only one equation from EH. Then, at each iteration, EH and PH are recalculated until a program P is found which covers E^{+} and whose optimality is greater than a desired value, provided by the user. To avoid infinite runs of the algorithm, it is limited to a maximum number of iterations which is also determined by the user. This all constitutes the function *StopCriterion*. Note that this stop criterion can be easily modified. For instance, it could be interesting to consider a cost-sensitive stop criterion based on the AUC of the programs.

The induction using background knowledge consists of performing the inverse narrowing operation with an equation from the background knowledge. The background equation is always taken as the sender rule. In this way, it is possible to obtain new equations with the background function on their rhs.

When the *InverseNarrowing* procedure does not generate new equations, the union of programs could be useful for adding non-recursive equations to the induced programs. These simple equations are usually the base case of recursive functions.

The parameters $factor\_union$ and $factor\_b$ determine the relative frequency of using union and using B respectively.

Next, we outline the IFLP algorithm.

```
Input: E+, E-, B, BF, factor_b, factor_union.
Output: a program P
begin
EH := ∅, PH := ∅, fb := factor_b,
fu := factor_union
GenerateCRG(input:E+, E-, ∅; output: EH)
PH := {{e}/e ∈ EH}
BestSolution := Select_best(PH)
while not Stop_Criterion(BestSolution) do

    round_robin(using_b, factor_b, fb),
    round_robin(using_union, factor_union, fu)
    if not using_b then begin

        n := card(E+)
        while n > 0 do

            PP := {(P1, P2) | P1, P2 ∈ PH and not marked[P1, P2], P1 ≠ P2
            s.t. card({e ∈ E+ | P1 |= e ∨ P2 |= e}) ≥ n}
            if PP ≠ ∅ then begin
              (P1, P2) := argmin_PP(Opt(P1) + Opt(P2))
              break while
            endif
            else n := n - 1

        endwhile
        marked[P1, P2]:= true
        BF' := ∅
    endif
    else begin

        P1 := Select_best(PH)
        P2 := B
        BF' := BF
    endelse
    InverseNarrowing(input:P1, P2, BF'; output:EH', PH')
    Update_all(BestSolution, EH, PH, EH', PH')
    if EH = EH' then begin

        if using_union then begin
         PH' := PH ∪ {P1 ∪ P2}
         if Select_best(PH) = Select_best(PH') then begin
                P = Select_best(PH)
                halt
                endif
         else Update_all(BestSolution, EH, PH, EH', PH')
         endif

    endif
endwhile
endalgorithm
```

where:

- $Select\_best(PH)$ selects the program with the best covering, and, finally, the best optimality.
- $Update\_all(S, E, P, E', P')$ performs the following actions:
  - $E = E \cup E'$ and $P = P \cup P'$
  - $S = Select\_best(P)$
- $marked[P, P']$ is true if P and P' have been selected to be combined by inverse narrowing in a previous step.
- $round\_robin(A, f, f')$ is defined as:

if $f' < 1$ **then begin**

$A = false$
$f' = f' + f$

end
else begin

$A = true$
$f' = f' - 1 + f$

endelse

The purpose of the matrix *marked* is to avoid the combination of rules that have been previously used. Note that the *round_robin* function allows to set or unset the factor A depending on the values f and f'.

The presented algorithm implements a learning framework, where evidence, programs and background knowledge are expressed using the same representation language: functional logic programs. This algorithm has important applications that will be discussed in Section 8.4. The main drawback of this algorithm is the low efficiency of learning process. Medium-size problems overflows the capacity of this system, mainly due to fact that the CRG operator generates a big quantity of rules just from one rule. In the next section we introduce an optimization of the present algorithm based on the use of incrementality.

7.5 Incremental Extension

As we have discussed previously, the IFLP algorithm can only deal with problems with few examples due to its low efficiency. In this section, we present a version of the IFLP algorithm in order to make it incremental, and in this way, to improve its efficiency.

Incrementality has previously been studied by the ILP and the ML community in general. Some incremental ILP systems are Clint [Raedt, 1992], Mobal [Kietz and Wrobel, 1992], Forte [Richards and Mooney, 1991] and Cigol [Muggleton and Buntine, 1992]. There are several reasons to use incremental systems. For instance, many problems can consist of a large evidence, which cannot fit in memory, and an incremental handling of this evidence is a straightforward and convenient solution. On the other hand, since induction is much more computationally expensive than deduction, incrementality permits the establishment of a hypothesis in the early stages of the learning process. If this hypothesis is stable, the next work will be deductive in order to check that the following evidence is consistent with the current hypothesis. On the other hand, predicates which are already known (learned) can be used as background knowledge in learning new predicates, and so on. This allows the learning of programs which define more than one concept at the same time in a way that is also incremental [Rao, 1997].

Given the current best hypothesis P selected by the algorithm, three possible situations can now arise each time that a new positive example e is presented:

Definition 7 (HIT) . *e is correctly covered by P, i.e., $P \models e$.*

Definition 8 (NOVELTY) . *Given the old evidence OE, a novelty situation is given when e is not covered by P but consistent, i.e., $P \not\models e \wedge \forall e' \in OE : P \cup \{e\} \models e'$.*

Definition 9 (ANOMALY) . *P is not positive consistent w.r.t.. $\{e\}$*

Both novelty and anomaly situations will require the revision of the current best hypothesis in order to match the new example. In the first case, the theory must be generalised in order to cover the new evidence, whereas in the other case, it must be specialised in order to eliminate the inconsistency. Hence, the topic known as theory refinement [Wrobel, 1993, 1996] is a central operation for incremental learning. The incremental reading of negative examples can also be contemplated, but, in this case, only hits and anomalies are possible.

In any incremental framework, the number of examples that should be read in each iteration must be specified. The most flexible approach is the use of adaptable values, which can be adjusted by the use of heuristics. The simplest case, on the contrary, is the use of a constant value, which is usually 1. In our case, we have adopted an intermediate but still simple approach. We use an initial incrementality value (*start-up* value) which is different from the next incrementality value. The reason for a different initial incrementality value is the CRG stage, which may require a greater start-up value than the next incrementality value (which is usually 1) in order for the main loop of the algorithm to begin with a meaningful population of programs.

According to the previous rationale, additional parameters are required in the new algorithm: i_0^+ is the initial positive incrementality value (*start-up* value), which determines how many positive examples are read initially. i_0^- is the initial negative incrementality value. i^+ is the positive incrementality value and i^- is the negative incrementality value, both of which determine how many positive and negative examples are read at each incremental step.

Plainly, the algorithm considers two cases. The first one is given when the best solution covers the new examples. In this case, the new examples are ignored and nothing else happens. This has been done in this way because if a right solution is found soon, the algorithm is highly accelerated as it only performs a deductive checking of subsequent examples.

In the case of a novelty or anomaly, the EH and PH sets are re-evaluated. The existing information for old evidence is reused, but the values are recomputed for the new examples. The result is that some equations and programs can be removed because they are inconsistent. The optimality of the elements of both EH and PH is recomputed from the old ones and the covering of the new examples. Then the best solution is obtained again in order to see whether there is a solution to the problem. In this case, nothing is done. Otherwise, the CRGs for the new examples are generated and added to EH and PH and the process continues.

This new algorithm allows for a much richer functionality for the IFLP framework. This approach can now accept a set PH of initial programs or theories P_1, P_2, ..., P_s, a background theory B and the examples. If these initial programs are not specified, the algorithm begins with no initial program set. If one or more initial programs are provided, it will build EH from all the equations that form the initial programs (avoiding duplicate equations) and will generate PH as initial programs plus a program for each equation from EH.

In this way, the framework can be applied for different tasks:

- a **pure inducer**: when there are no initial programs.
- a **theory reviser**: when a unique initial program is given. The program will be preserved until an example forces the algorithm to generate new equations and/or new programs.
- a **theory reviser/evaluator**: when several initial programs are given and n examples are provided with a (positive) incrementality value $i_0^+ < n$. In this case, the SelectBest function selects the best one w.r.t. the i_0^+ first examples. This program will be compared with subsequent examples and could be changed accordingly. In the end, the algorithm will indicate which initial program (or a new derived program) is the best w.r.t. the n examples.
- a **theory evaluator**: in this case, several initial programs are given and n examples are given with a value of (positive) incrementality $i_0^+ \geq n$. Then, the

optimality criterion is applied and the best program w.r.t. the n examples is chosen. Consequently, it simply indicates which of the initial programs is the best w.r.t. the evidence.

Initial theories, negative examples and background knowledge are optional for the incremental framework. The positive examples are also optional because the framework can also work as a theory evaluator for negative evidence only. Moreover, negative examples are not necessary for classification problems with a finite number of classes, due to the nature of functional logic languages. In any case, the parts of a theory which the user wants to be constant should be specified in the background knowledge.

7.6 Conclusions and Contributions of the Chapter

The IFLP paradigm extends the ILP paradigm similarly to the way in which functional logic programming subsumes logic programming. The operators and the algorithm which have been presented in this chapter successfully combine many different techniques from the FLP, ILP and ML communities: narrowing, approximations to termination, induction by generalisation, inversion of deductive operators, evolutionary techniques, and the MDL principle, giving a multipurpose, general and highly expressible paradigm as a result. The hypothesis language is restricted to unconditional equational systems with no extra variables on the rhs. The restriction of unconditional equational systems for the hypothesis language can only be considered to be a syntactical sugar restriction. By the use of the function $if(true, X) = X$, any conditional clause can be expressed as a non-conditional one. Consequently, the hypothesis language allowed by the framework subsumes that of ILP systems (except for the restriction of no extra variables on the rhs.) The hypothesis language permits more than one base clause (there is no implicit or explicit schema), there can be more than one recursive equation per function and there can be more than one recursive call per clause. Only a few systems of ILP (such as SKILIT [Jorge and Brazdil, 1995], PROGOL [Muggleton, 1995], FOIL [Quinlan, 1990b],...) have many of these salient features. However, they do not enjoy the easiness of defining functors and treating constructors that functional logic programming naturally deal with.

Chapter 8

The FLIP System

8.1 Introduction

In this chapter we present the system FLIP. The FLIP (Functional Logic Inductive Programmer) system is a realisation of the IFLP framework which was presented in the previous chapter. FLIP is a multiple function learner which is able to handle recursion and conditional background knowledge. It also uses both coverage and simplicity criteria to guide the choice of inverse narrowing steps and to select the best hypothesis. This chapter analyses the most relevant issues of the implementation of FLIP. For a user's manual, see Appendix B.

In Section 8.2, we describe the FLIP system and discuss its functionality. Some experimental results are presented in Section 8.3 showing the applications and the speed-up obtained by the incremental extension. A comparison with other systems is also included.

The contents of this chapter have been published in [Ferri *et al.*, 2000a,c, 2001b]

8.2 Implementation of the FLIP System

For the implementation of the IFLP framework, we have chosen the ANSI C programming language, because it allows us to deal easily with the great number of equations and programs that we generate during the induction process. System speed and portability are also reasons for this choice. In this section, we comment on some relevant aspects about the implementation of the Inverse Narrowing operator as well as the options of the system.

As we have seen in Section 7.4, the IFLP algorithm starts by generating a set of CRG's from the positive evidence. A program is built with each single CRG. The coverage and optimality of these programs are computed. These parameters are useful for selecting the programs to be combined by inverse narrowing. In particular, the selection of the best pair of programs to which inverse narrowing is applied is made regarding the coverage of the union of each pair of programs. Nevertheless, computing the coverage of every possible pair of programs could be very expensive because a narrowing derivation could be needed for each program and example. In this way, if there exist n initial programs, we would need to run C_2^n, that is, $(n^2-n)/2$ narrowing derivations for each example. Specifically, if there are m positive examples, it would perform $\mathcal{O}([(n^2-n)*m]/2)$ narrowing derivations in order to compute the coverage of every possible combination of two programs.

In order to avoid this expensive computation, we take an approach that consists in approximating the coverage of the new program as the disjunctive sum of the coverage of the pair of programs that form it. This approach will always be under the real value.

To deal with the coverage of a program, we use a binary array, where each bit indicates whether a positive example is covered. To compute the coverage of the union of two programs, it is enough to perform the binary OR of the corresponding arrays. With this solution, the selection of the programs to be combined has a complexity of $\mathcal{O}([(n^2-n)*m]/2)$ binary OR operations.

When each pair of programs is combined, the coverage of the program union is recorded. For this, the best structure is a triangular matrix. We denote this matrix as "map of programs". The purpose of the map of programs is to always have information about the existing programs and the combinations or crossings that have taken place, in order to select the pair to which inverse narrowing is to be applied.

The application of inverse narrowing to these programs generates a new set of programs. For each new program, its exact coverage and optimality are computed and a new row is added to the matrix, filling it in the same way as in the first step of the loop: calculating an approximate coverage of this new program and every other program in the matrix. In this way, the algorithm has a complexity of $\mathcal{O}([(n^2-n)*m]/2)$ the first iteration, and $\mathcal{O}(n*m)$ the following iterations, where n represents the number of programs, and m the positive examples.

The loop stops when either a program that covers all the examples is found or the algorithm has reached a limit of steps.

An important question to be determined before is how to choose the equations from the selected programs in every call of the inverse narrowing procedure. The algorithm uses two selection criteria: either to apply inverse narrowing to the equations with the greatest coverage of each program, in either direction; or to perform inverse narrowing between all the possible pairs formed by each equation of one program with all the equations of the other program, in either direction. Obviously, the second option (known as full inverse narrowing combination) produces a greater number of new equations.

The $InverseNarrowing$ procedure may generate equations that could be non terminating, for instance the equation $minus(X, Y) = -(minus(Y, X))$[1]. Equations of this kind would highly diminish the performance of the system whenever they would be sent to the narrowing solver. For this reason, we have implemented a simple filter that rejects those equations that need more than n narrowing steps to prove an example. This is just an approximation to the termination problem. A better way would be to implement a termination filter based on some well-known works about termination of CTRS [Arts and Giesl, 1996, 2000] in order to reject the greatest number of non-terminating equations as possible.

Finally, the current options of the system are:

- Termination Depth: Narrowing, the deductive operator which is used to compute the covering and consistency of any hypothesis, is bounded by a depth limit in the search tree. The shorter the value the more efficient the algorithm, but very short values can give incorrect results in certain cases if there are some large examples.

- Switch on/off the termination filter: In some cases it would be interesting not to reject the equations in which the termination filter does not hold. For example, the max function is induced with and without the termination condition. In the latter case, the clause $max(X, Y) = max(Y, X)$ is also generated. The same phenomenon occurs with the function $minus$, where the clause $minus(0, s(X)) = -(s(X))$ is replaced by $minus(X, Y) = -(minus(Y, X))$ if the termination filter is switched off. In some occasions, we may be interested in these algorithms, that, even though they are non-terminating, they do express general properties about the data.

- Full combination of inverse narrowing: This option should be set on for completeness. However, most problems can be solved without it and the execution times diminish considerably.

- Some other parameters: factor_b, factor_union and the maximum number of iterations.

The use of different values of the factor_union and factor_b parameters sometimes gives original results for the same problem. For instance, the if-then-else function is solved as:

$$ifthenelse(false, X0, X1) = ifthenelse(true, X1, X0)$$
$$ifthenelse(true, X0, X1) = X0$$

[1] $-$ is the constructor symbol used to represent the negative numbers.

with factor_union ≈ 0, or

$$\begin{aligned} ifthenelse(true, X, Y) &= X \\ ifthenelse(false, X, Y) &= Y \end{aligned}$$

with factor_union ≈ 1.

8.3 Experiments and Results

We tested our system with several examples of different kinds. FLIP usually finds the solution in a few loops, but, logically, this depends mainly on the number of clauses of the solution program and on the 'quality' of the initial examples. The length of the induced program is not limited. However, each new clause requires at least one iteration of the main loop. Consequently, FLIP deals better with shorter hypotheses.

8.3.1 Results Without Incrementality

Let us first study the non-incremental version of FLIP. Some non-recursive functional logic programs which were derived are shown in Table 8.3.1.

Induced program	Description
cup(y,up,X1,X2,n,y,X3)=true, cup(y,up,X1,X2,side,y,X3)=true	The Cup learning task where the arguments represent: BottomIsFlat, Concavity(up,not-up,no), Expensive, HandleOnTop,Fragile, Handle (n, top, side), Light, MadeOf(ceramic,paper,styrofoam)
enjoysport(sunny,X,Y,Z,W,V)=yes enjoysport(cloudy,X,Y,Z,W,V)=yes	The EnjoySport learning task where the arguments represent: Sky, Temp, Humidity, Wind, Water, Forecast
playtennis(overcast,X1,X2,X3)=yes playtennis(sunny,X1,normal,X3)=yes playtennis(rain,X1,X2,weak)=yes	The playtennis learning task where the arguments represent: Outlook, Temperature, Humidity, Wind
top(push(X,Y))=λ	Top element of a stack

Table 8.1: Some non-recursive programs induced without inverse narrowing.

The *cup*, *enjoysport* and *playtennis* learning tasks are extracted from [Mitchell, 1997]. The evidences which were used are those which appear in the book: 4 positive examples and 6 negative examples for the cup problem, 5 positive examples and 2 negative examples for the enjoysport problem and 9 positive examples and 5 negative examples for the playtennis problem. As could be expected, the CRG operator was sufficient for attribute problems of this kind. Hence, the program was induced before the first loop.

In the playtennis problem, the FLIP system generated a decision-tree through the exclusive use of the CRG operator (see Figure 2.2).

However, the main interest (and complexity) appears when learning recursive functions. In this sense, some relevant recursive functional logic programs induced without the use of background knowledge can be seen in Table 8.2. Again, functions such as *app* are more naturally defined as a function than as a predicate. No mode information is then necessary. Reversibility is not lost, as in the functional case, because, e.g., $app(X, a) = \cdot(b, \cdot(c, \cdot(a, \lambda)))$ returns $X = \cdot(b, \cdot(c, \lambda))$.

It should be highlighted that the FLIP system is not restricted to learning one function at a time, as can be shown for the *sum* & *prod* functions, which are induced together from mixed evidence.

Finally, more complex problems can be generated with the use of background knowledge, as illustrated in Table 8.3.

The results presented in this section were obtained by randomly selected examples, except those cases where the sources were identified. Evidences were relatively small in all cases: from 3 to 12 positive examples and from 2 to 11 negative examples. The values of the FLIP parameters used were factor_b=0.25, factor_union=0.5 and termination_depth = 12. For a more extensive account of results, examples used, etc, please see `http://www.dsic.upv.es/~flip/flip/`.

8.3.2 Speed-up Analysis

We have performed several other experiments with the incremental version of FLIP to learn well-known problems with and without knowledge. The inclusion of incrementality can highly improve the induction speed. Table 8.4 shows the speed-ups reached when inducing these problems running FLIP. Times were measured on a Pentium III processor (450 Mhz) with 64 MBytes of RAM under Linux version 2.2.12-20. They are expressed in seconds and are the average of 10 executions. The column Speed-up shows the relative improvement achieved by the incremental approach for $i_0 = 6$, obtained as the ratio Non-Incremental ÷ Incremental.

One might think that time could strongly oscillate depending on the start-up value. Nevertheless, as we illustrate in Figure 8.1, the use of incrementality generally improves induction time whereas the start-up value only affects the speed-up. Figure 8.1 expresses the learning time in seconds for different problems, depending on the start-up value[2].

[2]The points surrounded by a circle indicate experiments where the accuracy is below 100%.

Induced program	Description	Steps
sum(s(X),Y) = s(sum(X,Y)) sum(0,Y)=Y	Sum of two natural numbers	1
length(X,Y) = s(length(X)) length(λ)=0	Length of a list	2
consec(·(X,Y)) = consec(X) consec(·(·(X,Y),Y)) = true	Returns true if there exist two consecutive elements in a list	1
drop(0,X) = X drop(s(X),·(Y,Z)) = drop(X,Y)	Drops the N last elements of a list	1
app(·(X,Y),Z) = ·(app(X,Z),Y) app(λ,X) = X	Appends two lists	1
member(·(X,Y),Z) = member(X,Z) member(·(X,Y),Y) = true	Returns true if Z is in a list	1
last(·(X,Y)) = last(X) last(·(λ,X)) = X	Last element of a list	1
geq(s(X),s(Y)) = geq(X,Y) geq(X,0) = true	Returns true if the first element is equal or greater than the second	1
sum(s(X),Y) = s(sum(X,Y)) sum(0,Y)=Y prod(s(X0),X1) = sum(prod(X0,X1),X1) prod(0,X0) = 0	Addition and Multiplication at the same time	3
mod3(0) = 0 mod3(s(0)) = s(mod3(0)) mod3(s(s(0))) = s(s(mod3(0))) mod3(s(s(s(X0)))) = mod3(X0)	The mod 3 operation	3
even(s(s(X)) = even(X) even(0) = true	Returns true if a natural number is even	1

Table 8.2: Recursive programs induced without background knowledge.

The results show that increasing start-up values give increasing times, because induction (especially CRG's) is used more intensively. This is especially noticeable in the case of the *maxlist* problem, because the size of examples is large, and CRG's are very time consuming. Although lower i values give the most efficient results, there is also a risk of missing the good solution. This value is very dependent on the number of attributes. For instance, in the case of problems with 2 attributes, the optimal start-up value could be between 4 and 8 with some risk of missing the target program at low values. For the lenses problem (4 attributes) the optimal start-up value could be between 7 and 20. This suggests that the start-up value can be estimated from the number of attributes (which, moreover, is the only parameter that is known a priori).

Let us perform a more detailed study for a larger problem: the *monks1* problem. This is a popular problem from the UCI ML dataset repository [Blake and Merz, 1998] which defines a function such that $monks1(\_,\_,\_,\_,1,\_)$ and $monks1(X,X,\_,\_,\_,\_)$

Induced program	Description	Bkg.	Steps
$rev(\cdot(X0,X1)) = app(rev(X0),\cdot(\lambda,X1))$ $rev(\lambda) = \lambda$	Reversal of a list	append	2
$suml(\cdot(X0,X1)) = sum(suml(X0),X1)$ $suml(\cdot(\lambda,X0)) = X0$	Sum of a list of natural numbers	sum	1
$maxl(\cdot(X0,X1)) = max(X1,maxl(X0))$ $maxl(\cdot(\lambda,X0)) = X0$	Max of a list of natural numbers	max	1
$prod(s(X0),X1) = sum(prod(X0,X1),X1)$ $prod(0,X0) = 0$	Product of two natural numbers	sum	1
$fact(s(X0)) = prod(fact(X0),s(X0))$ $fact(0) = s(0)$	Factorial of a natural number	prod sum	4
$sort(\cdot(X0,X1)) = inssort(X1,sort(X0))$ $sort(\lambda) = \lambda$	Inefficient sort of a list	inssort, gt, if	1

Table 8.3: Some programs induced with background knowledge.

Benchmarks	Non-inc.	Inc.	Speed-up	#Rules	#Attributes
`sum`	6.01	0.42	14.30	2	2
`length`	13.96	5.51	2.53	2	2
`lenses`	12.17	2.15	5.66	9	4
`prod`	10.57	2.22	4.76	2 + 2 bkg	2
`maxlist`	39.66	2.27	17.47	2 + 3 bkg	2

Table 8.4: Results for $i_0 = 6$ and 24 examples. The accuracy for all of them is 100%.

are *true*. It has a larger number of examples to test and the function depends on 6 attributes. As expected, Table 8.5 shows that the speed-up increases as long as the number of examples increases. We have also measured the speed-up for different values of the start-up value for the *monk1* examples, ranging from 14 to 216. The times grow linearly from 400 s. to 4500 s.

8.3.3 Comparison of FLIP with other Systems

After these experimental results, we are going to make a comparison with other learning systems with regard to the power and expressibility of our inductive framework.

Apart from the use of different operators and the use of evolutionary programming techniques, the major differences between FLIP and other schema-less ILP systems lay in the representation language. According to the features of a comparison of ILP

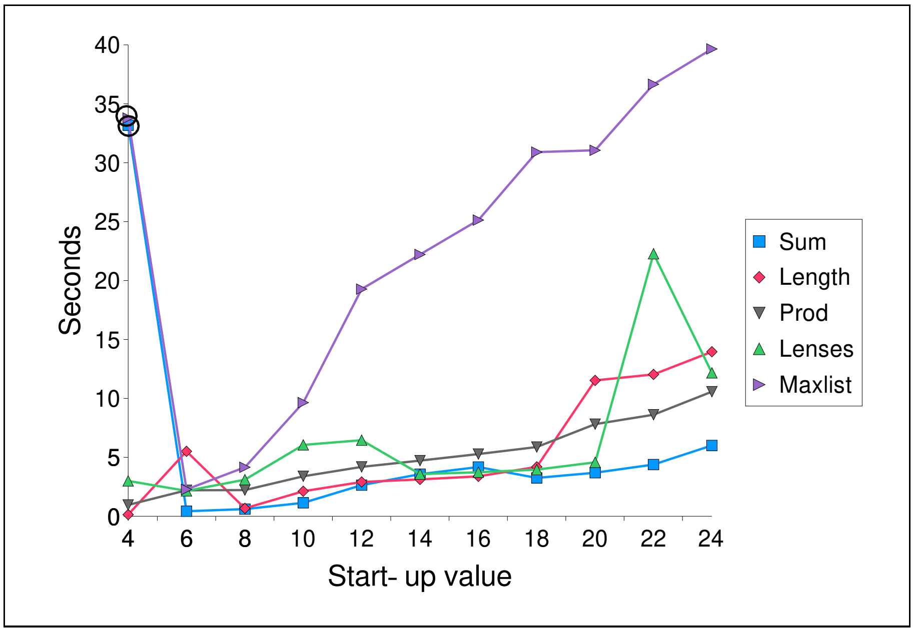


Figure 8.1: Times obtained in the induction of some problems depending on the start-up value.

systems which appeared in [Flener and Yilmaz, 1999], FLIP is based on a generalisation operator in a way similar to that which is done in some bottom-up ILP systems based on Plotkin's LGG's [Plotkin, 1971b,a]. In our case, we do not restrict FLIP to the most specific generalisation but allow all consistent generalisations and let the optimality criterion choose the most appropriate one in each case. In fact, FLIP learns from E^+ exclusively without the need for specialisation conditions, except for non-Boolean problems when negation is assumed to be negation as failure. FLIP also includes an operator which is inspired in the inversion of a deductive operator (like Muggleton's inverse resolution and inverse entailment [Muggleton, 1995]). In our case, the main operator for introducing recursion is the inverse narrowing operator.

The evidence language supported by FLIP consists of ground equations. These subsume ground literals, which are the most common evidence language used by ILP systems. The number of function symbols in the evidence is not restricted and FLIP can learn more than one symbol at a time. Moreover, FLIP learns from arbitrary and very reduced evidences. Type information is now not supported but its inclusion can only affect efficiency but not the power or completeness of the FLIP system. Mode information is not necessary for functions for obvious reasons and this makes the interaction with FLIP very friendly and easy to use. The user must only provide

Bench.	Non-inc.	Inc.	Speed-up	#Examples
monks1	528	345	1.53	25
monks1	1075	344	3.13	50
monks1	2012	347	5.8	100
monks1	3286	350	9.39	150
monks1	4598	351	13.10	216

Table 8.5: Benchmark results for $i_0 = 14$, and variable number of examples for the *monks1* problem. The accuracy for all of them is 100%. The number of rules is always 2.

	20%	40%	60%	80%	100%
FLIP	2—100%	2—100%	2—100%	2—100%	2—100%
C5.0	3—75%	7—91.67%	10—91.67%	10—91.67%	10—91.67%
C5.0-boosting	—81.94%	—96.7%	—97.45%	—100%	—100%
Rules	16—70.6%	15—96.7%	23—97.45%	16—100%	10—100%
TrainNet	68.5%	81.5%	100%	100%	100%
Progol	4—88.89%	9—94.44%	13—96.3%	13—98.14%	17—100%
FOIL	2—100%	2—100%	2—100%	2—100%	2—100%

Table 8.6: Learning table (number of rules of the program—accuracy) for different systems for the monks1 problem.

the positive (and possibly negative) examples in a file, the background knowledge if it exists and nothing more. In particular, the background knowledge can be any conditional functional logic program (any CTRS).

An experimental comparison of accuracy between different ML and ILP systems is illustrated in Tables 8.6 and 8.7. The tables show the accuracy and the number of rules of the program depending on the percentage of the training data employed. *C5.0* is a decision tree learner, evolved from C4.5 [Quinlan, 1993]. Boosting has been introduced by Quinlan in [Quinlan, 1996a]. *Rules* is a rule induction algorithm. *TrainNet* is an artificial neural network learning method. The three systems are part of the data-mining package Clementine 5.2 from SPSS [SPSS, 2002]. *Progol* v.4.4 [Muggleton, 2000] and *FOIL* v.6 [Quinlan, 1996c] are ILP systems. In the monks1 example, the accuracy given by FLIP was 100% from the very first training sample. In the lenses problem, only *Rules* behaved comparably to FLIP. Note that for the lenses problem *FOIL* is not able to learn a model due to the limited size of the evidence. Execution times are not shown since they are not comparable because the systems run on different platforms. FLIP is slower than the other systems, especially C5.0 and Rules which are faster by one or two orders of magnitude. Nonetheless, the difference

	20%	40%	60%	80%	100%
FLIP	2—70.3%	3—87.5%	5—87.5%	8—95.8%	9—100%
C5.0	2—70.3%	3—87.5%	3—87.5%	4—91.67%	4—91.67%
C5.0-boosting	—70.83%	—87.5%	—83.33%	—87.5%	—87.5%
Rules	2—70.3%	3—87.5%	6—91.67%	9—100%	9—100%
TrainNet	20.83%	64.585%	77.05%	68.75%	81%
Progol	5—20.83%	10—41.66%	14—58.33%	19—79.16%	24—100%
FOIL	-	-	-	-	-

Table 8.7: Learning table (number of rules of the program—accuracy) for different systems for the lenses problem.

in performance is highly reduced as the number of examples grows, due to FLIP's incremental treatment of examples.

8.4 Applications

The IFLP framework can be directly applied in the same fields where ILP has been successfully used [Muggleton, 1999]. Apart from the mining of the structured data, the IFLP framework is useful for other applications. Concretely, FLIP is being used for the automatic correction of functional-logic programs [Alpuente *et al.*, 2002, 2003; Correa, 2002] and in software engineering [Hernández and Ramírez, 2001].

With respect ILP, IFLP represents classification problems with more than 2 classes more naturally. Consider, for instance, the lenses problem from the UCI repository. Given four attributes of a patient, the goal is to predict which kind of lenses is required by patient from 4 attributes. The evidence of this problem for an ILP system, concretely PROGOL [Muggleton, 2000], is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mode declarations

  :- modeh(1,lens(+age,+spectacle,+astigmatic,+tear,-lenses))?
  :- modeb(1,lens(+age,+spectacle,+astigmatic,+tear,-lenses))?

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Types

  age(young).
  age(presby).
  age(prepresby).
  spectacle(myope).
```

```
  spectacle(hymyope).
  astigmatic(yes).
  astigmatic(no).
  tear(reduced).
  tear(normal).
  lenses(no).
  lenses(soft).
  lenses(hard).

% examples
  lens(young,myope,no,reduced,no).
  lens(presby,hymyope,no,normal,soft).
  lens(young,myope,no,normal,soft).
  lens(prepresby,hymyope,no,normal,soft).
  lens(young,myope,yes,reduced,no).
  lens(young,myope,yes,normal,hard).
  lens(young,hymyope,yes,normal,hard).
  lens(prepresby,hymyope,no,reduced,no).
  ....
```

First, it is required to indicate the modes of the literals, i.e. which attributes are parameters, and which ones are classes. Secondly, we need the types for all of them, even in this case types are not very significant to solve the problem. Finally, the examples are represented as facts.

Let us show the same evidence prepared to be used with the FLIP system.:

```
  lens(young, myope, no, reduced)= no,
  lens(young, myope, no, normal)= soft,
  lens(young, myope, yes,reduced)= no,
  lens(young, myope, yes, normal)= hard,
  lens(young, hymyope, no, reduced)= no,
  lens(young, hymyope, no, normal)= soft,
  lens(young, hymyope, yes, reduced)=no,
  ...
```

The examples are expressed as equations. No more information about the data is required. The use of equations makes it clear to see which is the class. The types of attributes are automatically induced by the system.

Other difference between ILP and IFLP is the dealing of two-class problems. In the ILP systems, one class is treated as positive examples while the other is employed for the negative evidence. In this way, an ILP system learns a program for the positive class. IFLP can consider two options, the first is to adopt the same configuration as ILP; in this way, a program is learned for the positive class; the other possibility is to express all the data as positive evidence. With this setting the IFLP system learns as program that returns the two possible classes without the necessity of a negative evidence.

Consider the *playtennis* example (see [Mitchell, 1997]); if we select the examples of the class *yes* as positive evidence and the examples of the class *no* as negative evidence, FLIP learns the following program:

```
playt(rain,X0,X1,weak) = yes
playt(sunny,X0,normal,X1) = yes
playt(overcast,X0,X1,X2) = yes
```

On the other hand, if all the data is placed at the positive evidence, FLIP learns a problem that consider the two classes:

```
playt(sunny,X0,high,X1) = no
playt(rain,X0,X1,strong) = no
playt(sunny,X0,normal,X1) = yes
playt(overcast,X0,X1,X2) = yes
playt(rain,X0,X1,weak) = yes
```

Another kind of data-mining application for which FLIP is advantageous is the mining of unstructured data, an area that is being increasingly more important. Most information in the web is unstructured or semistructured. In order to do this, learning tools should be able to cope with this kind of data. For instance, given an eXtended Mark-up Language (XML) document [Abiteboul *et al.*, 1999] which contains information about the good customers of a car insurance company:

```
<goodc> <name>john</name> <has_children/> </goodc>
<goodc> <married/> <teacher/> <has_cellularphone/> </goodc>
<goodc> <sex>male</sex> <teacher/> <name>jimmy</name> </goodc>
...
```

This format of data cannot be addressed by usual data-mining tools, because the data is not structured within a relation and also because the possible attributes are unordered and it can be of different number and kind for each example. Nonetheless, it can be processed by our system FLIP by automatically converting XML documents into functional terms trees. In this case, the resulting trees would be:

```
goodc(p(p(v,name(john)),has_children))=true
goodc(p(p(p(v,married),teacher),has_phone))=true,
goodc(p(p(p(v,sex(male),teacher),name(jimmy))))=true,
goodc(p(p(v,sex(female)),tall))=true,
```

```
goodc(p(p(v,nurse),sex(female)))=true,
goodc(p(p(p(v,browneye),likes_coffee),has_children))=true,
goodc(p(p(p(v,has_children),nurse),has_phone))=true,
goodc(p(p(p(v,name(jane)),plays_chess),has_children))=true,
goodc(p(p(p(p(v,has_children),sex(female)),name(joan)),speaks_spanish))=true,
goodc(p(p(p(p(v,name(jane)),sex(female)),speaks_portuguese),tall))=true,
goodc(p(p(p(p(p(v,name(jimmy)),has_children),teacher),sex(male)),tall))=true,
goodc(p(p(p(p(p(v,teacher),low_income),is_atheist),has_phone),married))=true,
goodc(p(p(p(p(v,name(mary)),speaks_spanish),has_children),has_phone))=true
```

and the evidence for the other class (negative evidence), extracted from bad customers:

```
goodc(p(p(v,sex(male)),tall))=true,
goodc(p(p(v,nurse),sex(male)))=true,
goodc(p(p(v,name(peter)),married))=true,
goodc(p(p(p(v,married),policeperson),has_phone))=true,
goodc(p(p(p(v,name(charlie)),sex(male)),butcher))=true,
goodc(p(p(p(v,browneye),likes_coffee),sex(male)))=true,
goodc(p(p(p(v,plays_football),nurse),has_phone))=true,
goodc(p(p(p(v,susan),plays_chess),married))=true,
goodc(p(p(p(p(v,butcher),sex(male)),name(paul)),speaks_spanish))=true,
goodc(p(p(p(p(v,name(steve)),sex(male)),speaks_portuguese),tall))=true,
goodc(p(p(p(p(p(v,name(pat)),speaks_french),mad),sex(male)),high_income))=true,
goodc(p(p(p(p(p(v,policeperson),low_income),atheist),has_phone),married))=true
```

then, FLIP gives the following solution to it:

```
goodc(p(X0,X1)) = goodc(X0)
goodc(p(X0,has_children)) = true
goodc(p(X0,sex(female))) = true
goodc(p(X0,teacher)) = true
```

Note that the solution is recursive and hence can deal with this unstructured data.

8.5 Limitations of the IFLP approach

In the previous section we have shown some applications where the use of a functional logic language as representation language makes it possible to deal in a natural way with problems with a complex structure. However, the current approach presents some limitations that can make the use of FLIP system difficult for learning in some application areas.

The CRG operator is a powerful method to construct equations from an example which are consistent with respect to the positive and negative evidence. However, the number of equations created grows exponentially with the number of attributes of the example. Therefore, the learning of problems with a great number of attributes cannot be performed by FLIP under reasonable resource limitations.

Although the incremental version of the algorithm has extended the applicability of the system, the induction of hypotheses from problems with a huge amount of examples is not possible, due to the great number of rules generated by the operators. This is also a problem of many ILP systems.

Other important limitation in the current version of FLIP is the impossibility of dealing with numerical attributes. This is mainly because of the absence of mechanisms to evaluate numerical variables in basic functional logic languages. This limitation could be overcome by the incorporation of methods for supporting numerical values in the IFLP algorithm. For instance, [Lux, 2001] defines a method to support real numbers in Curry by linear constraints.

The presence of noisy data is not actually considered in the learning process. Therefore, the framework tends to overfit the training data.

Finally, other problem that strongly affects the performance of the FLIP system is the appearance of non-terminating rules or programs during the learning process. In the current version there is a simple filter that eliminates some of these programs. It could be very useful to improve this filter by modern and effective techniques of termination analysis. However, it must be realised that a complete filtering of the non-terminating programs is impossible because termination of TRS is undecidable [Dershowitz, 1987]. Note that this problem is inherent to systems that can learn recursive programs.

8.6 Conclusions and Contributions of the Chapter

The course from theory to implementation illustrated in the IFLP paradigm is guided by a compromise between accuracy and efficiency. The FLIP system represents a proper example of the outcome of this compromise. FLIP is able to induce attribute problems (concept learning), classification problems, decision trees, recursive programs, conditional theories, multiple function problems at the same time and can exploit the information expressed in the background knowledge. Some specific problems are advantageously addressed by the IFLP paradigm, such as classification problems for relational and non-relational (semi-structured) data. FLIP is not specialised for any of these particular problems, but it behaves reasonably well for each of them.

Due to the IFLP paradigm, the FLIP system has all the advantages of the expressiveness of the ILP systems as well as a more natural representation for classification problems, which are usually given in a functional way. This does not preclude the use of non-injective functions, which can also be dealt with the use of the narrowing operator, which performs unification instead of matching and can return results through the variable substitutions as is done in logic programming.

With regard to scalability, the results which were presented initially were obtained with a very small number of iterations and a few seconds on a personal computer. The incremental version has permitted the handling of more complex problems and datasets with a medium number of examples, widening the applicability of our framework and system. However, there are some important limitations that prevent to apply FLIP system for problems with a great amount of data or/and attributes, or with numerical attributes, which are typical in many data-mining applications.

The following chapters propose an alternative paradigm to overcome these problems.

Part III

Decision Multi-trees

Chapter 9

Decision Multi-tree Learning

In the previous chapters we have introduced a framework for the learning of functional logic programs. This framework presents several advantages to address problems when the representation language can play an important rule. Problems with an inherent complex structure can be elegantly expressed into a functional logic framework. However, the Achilles' heel of this framework (and of most relational learning approaches) is mainly the low efficiency of the learning algorithm. This deficiency precludes the employment of relational learning in many real-world problems. In these cases, the evidence is formed by thousands or even millions of examples and, more importantly, dozens or even hundreds of attributes. This great quantity of data usually overflows the capacity of many relational learners.

One usual way to improve the efficiency of the learning techniques is to restrict the expressiveness of the representation language in order to reduce the search space. For this reason, propositional algorithms can deal with a huge amount of evidence due to the simple representation language (propositional rules) that they employ. From all the efficient propositional methods, decision tree learning represents a popular method due to the quality and comprehensibility of the models generated by this method. However, decision trees are usually too eager, and sometimes the user would like to invest more time in order to obtain a better hypothesis.

In this chapter, we present a method that allows a set of decision trees to be generated from a single evidence. These decision trees can be combined to give a highly accurate (but complex) model. Additionally, one (or more) single decision trees can be extracted and still represented as propositional rules. The work presented in this chapter has been published in [Ferri *et al.*, 2001a,c, 2002c].

9.1 Introduction

Chapter 2 has reviewed decision-tree learning, and in Chapter 6 we saw ensemble methods as a technique for generating multiple decision trees and combining them. All of them are based on a collection of trees, which is called a forest. Our approach is based on a shared ensemble, a group of decision trees that shares the common parts what we call *decision multi-tree*. The decision multi-tree learning method is based on classical decision-tree learning; more precisely the idea is to generate an AND/OR tree structure called multi-tree, from which it is possible to extract a set of hypotheses. We perform a greedy search for each solution, but once the first solution is found the following ones can be obtained taking into consideration a limited computation time. Therefore, our algorithm can be considered *anytime* in a certain way. An anytime algorithm is one that can be interrupted at any point during computation to return a result whose quality increases with increasing computation time [Dean and Boddy, 1988].

As we have discussed, the multi-tree structure contains a collection of decision trees. Given this set of hypotheses, we can study a criterion to select the best one in accuracy and comprehensibility, or we can combine all of them obtaining an even more accurate model but, unfortunately, not comprehensible. In this chapter we concentrate on the construction of decision multi-trees, and then, how to extract one (or more) models that will be accurate and comprehensible. The use of multi-tree learning as an ensemble method and some other researches will be addressed in the next chapter.

9.2 Multi-tree Structure

Usually, as we have described in Section 2.2, the construction of a decision tree is performed in two different steps [Quinlan, 1993]:

- **Tree Construction:** The entire decision tree is constructed in this phase. The process is driven by a splitting criterion that selects the best split. The selected split is applied to generate new branches, and the rest of the splits are discarded. The algorithm stops when the examples that fall into a branch belong to the same class. In this case, the branch is considered a leaf, and it is not further explored.

- **Pruning:** This phase consists in the simplification of some parts of the tree to avoid overfitting. There are two options: *pre-pruning*, when the process is performed during the construction of the tree, or *post-pruning*, when the pruning is performed by analysing the leaves once the tree has been built.

Thus, decision trees are built in an eager way, which makes them suitable for the quick construction of a model. However, this may produce inaccurate models because of bad decisions in the split selection.

In order to make a more rational and anytime search, we have derived a new structure from which more than one model can be obtained. We have called this structure *decision multi-tree*. Basically, a decision multi-tree is a tree in which the rejected splits are not removed, but stored as *suspended* nodes. The further exploration of these nodes after the first solution is built permits the generation of new models. For this reason, we call this structure decision multi-tree, rather than a decision tree. Since each new model is obtained by continuing the construction of the multi-tree, these models share their common parts.

Likewise, a decision multi-tree can also be seen as an AND/OR tree [Nilsson, 1998; Pearl, 1985], if one considers the split nodes as being OR-nodes and considers the nodes generated by an exploited OR-node as being AND-nodes.

Formally, a decision multi-tree is formed by an AND-node at the root, with a set of children which are OR-nodes (each one represents the splits considered). Each OR-node can be *active* or *suspended*. The active OR-nodes have a set of children which are AND-nodes corresponding to a descendant of the split. This schema is repeated for each new descendant node (AND-node), until the AND-node contain examples of just one class (a leaf). This way of exploring the space in order to generate an ensemble of hypotheses can be seen as a beam search [Nilsson, 1998].

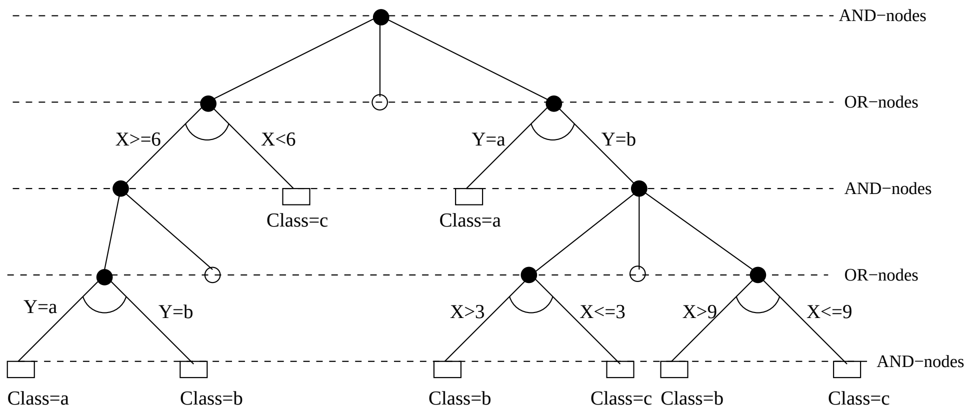


Figure 9.1: The multi-tree structure.

Figure 9.1 shows a decision multi-tree. OR-nodes are represented with an arc and leaves are represented by rectangles. The active OR-nodes are black, while the suspended ones are white. In this multi-tree five OR-nodes have been explored. In a decision multi-tree, as we can see, there are alternatively levels of AND-nodes and

OR-nodes. Note that a classic decision tree can be seen as a decision multi-tree where only one OR-node is explored at each OR-node level.

Three different models can be obtained from this multi-tree:

Decision Tree 1:

`if` $(X < 6)$ `then`	`Class` $= c$
`else if` $(Y = a)$ `then`	`Class` $= a$
`else if` $(Y = b)$	`Class` $= b$

Decision Tree 2:

`if` $(Y = a)$ `then`	`Class` $= a$
`else if` $(X > 3)$ `then`	`Class` $= b$
`else`	`Class` $= c$

Decision Tree 3:

`if` $(Y = a)$ `then`	`Class` $= a$
`else if` $(X > 9)$ `then`	`Class` $= b$
`else`	`Class` $= c$

For the construction of a decision multi-tree it is necessary to specify two new criteria apart from the two required for the construction of a single decision tree (see Section 2.2.1). These new criteria are:

- **Suspended OR-node selection:** Selects one of the suspended nodes from where the population of the multi-tree is continued. Note that the first OR-node at each OR-node level is selected by the splitting criterion, while this criterion selects among the suspended OR-Nodes, once a first hypothesis has been constructed.

- **Selection of models**: Since a multi-tree represents a set of models, in order to select just one single model we need to define a criterion to extract just one from the whole set of models in the multi-tree.

The decision multi-tree approach presents some interesting features. First, the number of solutions grows exponentially with respect to the number of suspended OR-nodes that are exploited. Secondly, the solutions share some of their parts, so this exponential increase in alternatives only requires a sublinear increase in resources. The percentage of the shared quantity depends on the depth of the suspended OR-nodes that are exploited. Exploring suspended nodes at the bottom areas of the multi-tree causes the generation of models that share many of their conditions; therefore they must be very similar. However, the exploration of OR-nodes at the top positions of the multi-tree produces solutions that might be different enough, but they share few conditions. For instance, the decision trees 2 and 3 of Figure 9.1 share the split $Y = a|Y = b$.

To know how many models there are in a multi-tree, we can follow this method:

1. A value named NM, which is initialised to 1, is assigned to each leaf node
2. This value is propagated upwards the multi-tree as follows:
   (a) For the OR-nodes $NM = NM_1 \cdot NM_2 \cdot NM_n$, where NM_i is the NM value of the i-th child of that OR-node.
   (b) For AND-nodes $NM = NM_1 + NM_2 + \ldots + NM_n$, where NM_i is the NM value of the i-th child of that AND-node.
3. When the propagation reaches to the root of the multi-tree, NM indicates the number of solutions in the multi-tree.

If we apply this method to the multi-tree of Figure 9.1, the result is 3, as expected. Figure 9.2 shows the NM value of each node.

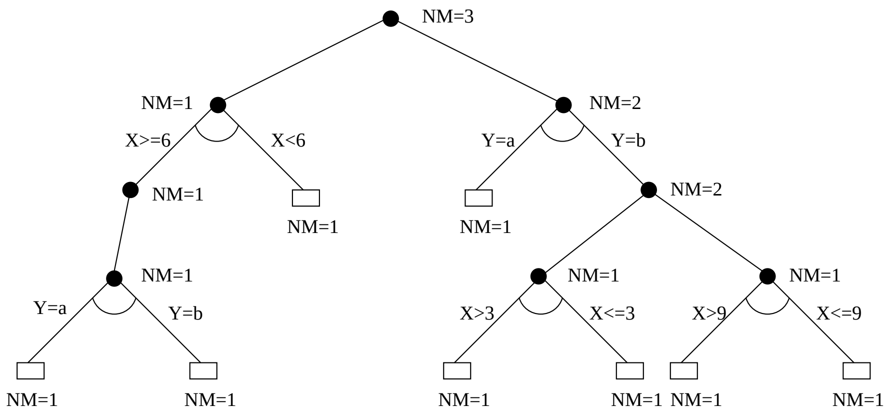


Figure 9.2: Propagation of NM value.

In the following subsection, we discuss about some criteria to populate the multi-tree, and then, to select one or more models from the multi-tree structure.

9.2.1 Suspended OR-node Selection Criterion

The population of a decision multi-tree is performed by selecting one suspended OR-node from the list of suspended OR-nodes. To select this node, a selection criterion must be defined. As we have said before, we have called this criterion the *suspended OR-node selection criterion*. The *optimality* (see Section 2.2.1) can also be used to choose the OR-node to be explored. We have considered the following suspended OR-node selection criteria:

- **Rival absolute:** From the list of suspended OR-nodes, the node with highest optimality is selected.
- **Rival Ratio:** For every suspended OR-node, a rival ratio value is computed, as the relation between the *optimality* of the OR-node and the best *optimality* of its sibling OR-nodes (i.e the children of the same AND-node). The suspended OR-node with highest rival ratio value is chosen.
- **Rival Ratio with Depth Weighted:** This criterion is identical to the rival ratio criterion, but in this case we add to the split ratio value a factor that depends inversely on the depth of the OR-node.
- **Random:** This criterion just selects nodes pseudo-randomly (uniform distribution) from the list.
- **Random with depth weighted:** This criterion just selects nodes in a pseudo-random way but taking into account the depth of the selected node.
- **TopMost:** Select first the topmost node with highest optimality.

We will analyse experimentally their behaviour in Section 9.3.1.

9.2.2 Selection of Models from the Multi-tree

After the exploration of a given number n of suspended-OR nodes, the multi-tree will contain an exponential number of hypotheses. Therefore, the following step is to establish a strategy to extract one or more solutions from the multi-tree.

Occam Selection

The first and most simple option that we have considered is the extraction of the simplest model, i.e. the model with the fewest number of rules. This is based on the Occam principle that consists in preferring simpler solutions over larger ones.

To select the simplest solution, once the multi-tree is completely constructed, we assign a variable called NR to each node, and then we traverse it from bottom nodes to the root node, in the following way:

1. NR is initialised to 1 in the leaf nodes.
2. This value NR is propagated upwards the multi-tree:

(a) In the OR-nodes $NR = NR_1 + NR_2 + \ldots\ NR_n$, where NR_i is the NR value of the i-th child of that OR-node.

(b) In the AND-nodes, $NR = \min(NR_1, \ldots, NR_n)$, i.e. we assign the child OR-node with the minimum NR value. This OR-node is also marked in order to recover the shortest solution later.

3. When the propagation reaches the root of the multi-tree, NR contains the minimum number of rules of a solution in the multi-tree. To get this solution, we only have to follow the marked OR-nodes, at the OR-nodes level.

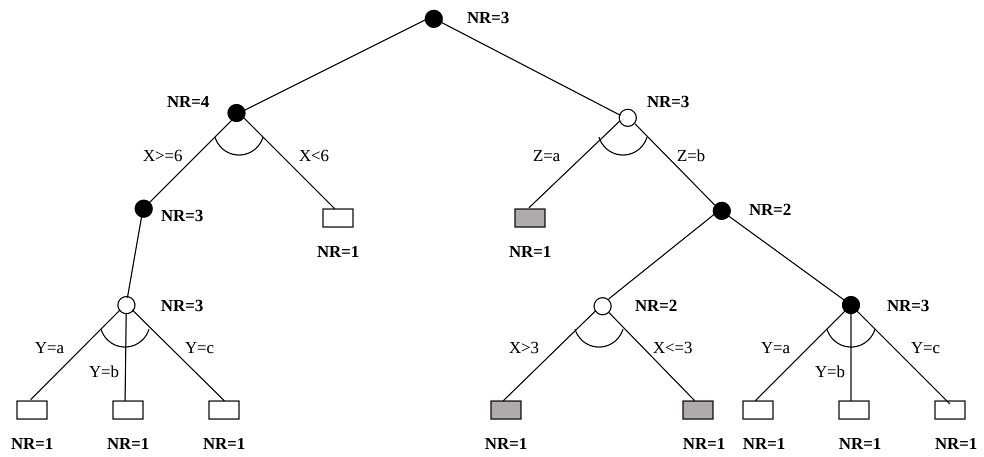


Figure 9.3: Selection of the simplest decision tree from the multi-tree structure.

Figure 9.3 shows the selection of one hypothesis from a multi-tree according to the number of rules. The OR-nodes are represented with an arc. The leaves are represented by rectangles. First, we fill the leaves. Then, we propagate the NR value upwards as has been previously detailed. Finally, when we reach the top of the tree, it is straightforward to extract the solution just descending the multi-tree by the marked OR-nodes. In the figure, the marked OR-nodes are represented by the white circles, and the leaves of the selected hypothesis are shadowed.

Selection of s-best different solutions

The previous procedure can be employed to select just one hypothesis from the multi-tree. Nevertheless we can also extract more hypotheses from the multi-tree. In some scenarios it is very interesting to offer several comprehensible models with similar accuracy; in this way the user can select the one that she prefers. This is the case, for instance, of some medical applications, where physicians appreciate that the system could offer several models very helpful to assist her final decision. Diversity among this set of produced models is also another important factor to be taken into account in these cases. The following method provides a way to obtain a set of diverse solutions based on the Occam principle. The procedure to obtain s solutions is described as follows:

1. Two variables UNUSED-ORS and VISITS are created in each node. The first one is initialised as the number of suspended OR-nodes that fall under a particular node. The second one indicates the number of times that the node has been employed to construct a solution. Initially, it is initialised to 0 for each node, except from the nodes that are in the first solution; in that case it is initialised to 1.
2. Initialise a global counter of solutions: $c = 0$.
3. **While** $c < s$ **do**.
   (a) If the number of UNUSED-ORS of the root equals zero then **stop**.
   (b) Call *Explore-AndNode(root)*.
   (c) Store the solution S_c, increment c.
4. EndWhile.

Where the function *Explore-AndNode()* is defined as follows:

Explore-AndNode(AND-node)

1. If the current node is a leaf then add the leaf to solution S_c and **return**.
2. Otherwise, it has as children an OR-node alternative. Choose one according to this criterion: from all the children where UNUSED-ORS > 0 then select the one with the lowest number of VISITS. If there is a tie, select the one with the highest solution optimality.
3. Select this node and increment by 1 its VISITS and decrement by 1 the variable UNUSED-ORS of all the nodes which are parent of this node until the root node.
4. Call *Explore-AndNode(AND-node)* for all the AND-nodes.

This eases the selection of quite different solutions (provided the multi-tree has these solutions), because visited branches are penalised until there is no other option, and unvisited OR-nodes are sooner or later explored. With equal conditions, optimal OR-nodes are explored first, so ensuring a reasonably good accuracy of the different s solutions.

Example 13 *Consider the* playtennis *example [Mitchell, 1997]. If we execute the previous algorithm with s=3, without pruning, and GainRatio [Quinlan, 1993] as splitting criterion, we obtain these three different solutions.*

Hypothesis 1:

```
f(overcast, X1, high, X3)   =  yes
f(rain, X1, high,weak)      =  yes
f(rain, X1, high, strong)   =  no
f(sunny, X1, high, X3)      =  no
f(X0, X1, normal, X3)       =  yes
```

Hypothesis 2:

```
f(overcast, X1, X2, X3)   =  yes
f(rain, X1, X2,weak)      =  yes
f(rain, X1, X2, strong)   =  no
f(sunny, X1, high, X3)    =  no
f(sunny, X1, normal, X3)  =  yes
```

Hypothesis 3:

```
f(overcast, X1, high,weak)     =  yes
f(rain, X1, high,weak)         =  yes
f(sunny, X1, high,weak)        =  no
f(X0, X1, normal, weak)        =  yes
f(overcast, X1, X2, strong)    =  yes
f(rain, X1, X2, strong)        =  no
f(sunny, X1, high, strong)     =  no
f(sunny, X1, normal, strong)   =  yes
```

All the three hypotheses[1] *have 100% of accuracy, however they differ in the number of rules and/or in the conditions tested. Thus, they can be interpreted as three different solutions for the same problem.*

In comparison with the IFLP learning method seen in the previous chapters, decision multi-tree learning has very different features: first, the representation language is now propositional, therefore the algorithm is very limited in the learning from problems with complex structure and in the use of background knowledge. On the other hand, since the search space is more limited, the multi-tree is constructed more quickly, and it can be employed for medium or even large problems. Finally, the multi-tree can generate a set of models from the same evidence, while IFLP only produces one.

[1]The hypotheses are presented as functional logic programs derived from the decision trees.

9.3 Experimental Evaluation

In this section we are going to evaluate experimentally the ideas proposed in this chapter. The methods and algorithms presented in this chapter have been included in the SMILES system [Ferri *et al.*, 2002d] (see Appendix C).

For the experiments we employ 28 datasets extracted from the UCI repository [Blake and Merz, 1998]. A detailed description of the datasets can be found in Appendix A.

The decision multi-trees are constructed by using GainRatio [Quinlan, 1993] as splitting criterion. For a better evaluation, we have performed the experiments by averaging 10 results of a 10-fold cross-validation. This makes a total of 100 runs (each one with a different multi-tree construction and hypothesis selection process) for each pair of method and dataset. Pruning is not enabled. The experiments were executed on a Pentium III 733 Mhz with 512 MB of memory running Linux version 2.4.7.

9.3.1 Suspended OR-node Selection Criterion

Let us study the different strategies to populate the multi-tree. For this purpose we compute the accuracy of the Occam solution of a multi-tree constructed by exploring 100 suspended nodes according to the following strategies: bottom, rival absolute, rival ratio, random and topmost. They are compared with one decision tree (first solution) constructed in the original way [Quinlan, 1993]. Table 9.1 shows the accuracy and Table 9.2 shows the AUC of each dataset and the geometric mean of all the datasets. Table 9.3 presents the average learning time (in seconds) and the geometric mean of all the datasets.

The results demonstrate that the best results in terms of accuracy for the Occam solution are obtained by the rival absolute and the rival ratio (0.5% more in accuracy). Regarding AUC, the strategies obtain very similar results, only rival ratio seems to improve slightly the AUC measure. However, rival absolute requires much less time than rival ratio. Hence, rival absolute seems to be the best strategy to populate the multi-tree when the Occam strategy is employed to extract a solution from the multi-tree.

9.3.2 Extraction of a Hypothesis from the Multi-tree

In this subsection we study the strategy to extract a solution from the multi-tree. This strategy is based on the Occam principle.

To build the multi-tree, we employ the rival absolute suspended OR-node selection criterion, which according to the previous experiments seems to obtain good results.

Table 9.4 shows the evaluation in accuracy of the Occam solution depending on the size of the multi-tree. In general, the results indicate that the accuracy of the

#	1st	Bottom	Rival Abs.	Random	Rival Ratio	Top Most
1	77.94	77.69	77.68	77.55	77.63	77.69
2	93.81	93.81	93.99	93.49	93.45	93.78
3	92.43	93.39	92.48	92.77	93.75	92.86
4	88.61	89.00	88.92	88.87	89.01	88.83
5	99.61	99.61	99.64	99.64	99.64	99.61
6	48.29	48.29	48.37	47.90	48.31	44.69
7	91.47	91.44	92.31	92.69	92.17	94.08
8	38.23	39.62	40.23	38.15	35.85	36.54
9	66.33	66.30	66.47	65.70	65.37	64.77
10	74.06	73.25	73.06	72.94	73.06	73.06
11	52.08	51.98	51.32	51.89	52.20	53.80
12	75.80	75.80	76.13	74.20	75.93	74.40
13	62.72	62.72	62.64	62.58	62.69	66.39
14	78.53	78.53	78.53	78.11	78.33	79.17
15	94.74	94.74	94.65	94.21	94.35	93.14
16	94.13	94.07	94.13	94.47	94.20	94.73
17	64.44	64.50	64.21	64.32	65.29	69.71
18	95.22	99.00	98.31	99.80	99.73	100.00
19	70.97	71.17	71.22	69.98	71.05	69.08
20	97.47	97.67	97.67	97.56	97.38	97.47
21	91.95	92.38	92.43	93.14	93.14	95.14
22	69.63	69.62	69.51	69.16	71.71	69.62
23	62.25	66.13	67.00	63.00	69.88	62.38
24	95.91	95.90	95.98	96.15	96.06	96.06
25	60.60	60.53	61.27	60.47	60.07	60.07
26	99.23	99.23	99.31	99.26	99.23	99.38
27	77.16	77.53	77.74	77.96	78.13	77.43
28	93.00	93.06	93.76	93.35	94.00	93.59
GMean	76.73	77.13	77.22	76.75	77.16	76.95

Table 9.1: Accuracy of the Occam solution in the multi-tree depending on the suspended OR-node selection criterion.

#	1st	Bottom	Rival Abs.	Random	Rival Ratio	Top Most
1	0.824	0.823	0.822	0.827	0.823	0.820
2	0.979	0.979	0.979	0.974	0.977	0.974
3	0.968	0.656	0.968	0.971	0.971	0.984
4	0.953	0.951	0.947	0.951	0.952	0.952
5	1.000	1.000	1.000	1.000	1.000	1.000
6	0.657	0.639	0.657	0.654	0.657	0.625
7	0.991	0.991	0.992	0.994	0.992	0.995
8	0.622	0.634	0.639	0.616	0.624	0.622
9	0.641	0.666	0.640	0.639	0.631	0.654
10	0.900	0.874	0.872	0.873	0.872	0.872
11	0.667	0.718	0.660	0.667	0.663	0.678
12	0.718	0.724	0.721	0.741	0.746	0.746
13	0.724	0.823	0.724	0.713	0.722	0.775
14	0.823	0.579	0.824	0.824	0.820	0.834
15	0.987	0.987	0.986	0.983	0.985	0.976
16	0.987	0.987	0.987	0.989	0.988	0.988
17	0.667	0.667	0.666	0.668	0.669	0.711
18	0.987	0.995	0.997	1.000	1.000	1.000
19	0.710	0.707	0.713	0.682	0.698	0.662
20	0.991	0.992	0.990	0.990	0.989	0.990
21	0.970	0.970	0.969	0.974	0.976	0.983
22	0.761	0.761	0.760	0.756	0.789	0.761
23	0.559	0.998	0.582	0.558	0.635	0.547
24	0.997	0.763	0.997	0.997	0.997	0.997
25	0.763	0.872	0.761	0.760	0.756	0.759
26	0.998	0.997	0.999	0.998	0.998	0.999
27	0.873	0.976	0.874	0.880	0.879	0.875
28	0.976	0.976	0.978	0.975	0.979	0.973
GMean	0.833	0.834	0.834	0.831	0.837	0.835

Table 9.2: AUC of the Occam solution in the multi-tree depending on the suspended OR-node selection criterion.

#	Bottom	Rival Abs.	Random	Rival Ratio	Top most
1	0,05	0,06	1,10	0,08	31,41
2	0,81	0,34	2,47	0,29	21,92
3	0,09	0,56	11,86	8,77	136,38
4	0,05	0,09	0,43	0,10	1,47
5	0,39	0,63	5,78	0,91	0,30
6	0,56	0,46	5,64	40,11	939,99
7	0,30	0,33	3,51	1,09	16,03
8	0,12	0,08	0,80	0,17	3,27
9	0,04	0,05	0,98	3,93	34,88
10	0,04	0,02	0,07	0,03	0,03
11	0,40	0,36	5,00	5,64	255,20
12	0,06	0,07	1,52	0,13	7,92
13	0,21	0,18	3,59	0,58	83,75
14	0,17	0,14	3,19	0,26	41,93
15	0,02	0,08	0,54	0,07	2,62
16	0,03	0,12	0,35	0,05	1,59
17	0,01	0,17	6,10	135,59	76,48
18	0,16	0,02	0,16	0,04	0,26
19	0,02	0,03	0,29	0,07	1,05
20	0,03	0,02	0,15	0,02	0,18
21	0,01	0,16	0,91	0,96	6,93
22	0,03	0,76	12,26	147,85	0,81
23	0,01	0,02	0,21	0,03	0,48
24	1,32	2,71	15,92	28,78	15,82
25	0,03	0,04	0,85	11,40	118,58
26	1,24	2,17	11,31	7,56	14,62
27	0,05	0,05	0,46	0,06	9,02
28	0,06	0,57	1,99	0,68	880,00
GMean	0,09	0,14	1,43	0,68	8,55

Table 9.3: Average learning time (in seconds) depending on the suspended OR-node selection criterion.

Occam solution is increased when the multi-tree is further populated.

Regarding the AUC value, Table 9.5 includes the results. The Occam solution also improves the quality of the classifiers with respect to the first solution (a simple decision tree); nevertheless, in this case, the differences are very small.

Lastly, the comprehensibility of the solutions is also slightly improved, because the number of rules of the model is reduced. Table 9.6 contains the average number of rules of each dataset and the geometric mean of all the datasets. Obviously, when the size of the multi-tree is increased, shorter solutions are generated and hence can be found.

9.4 Conclusions and Contributions of the Chapter

In this chapter, we have presented an algorithm that has been designed to give multiple solutions in an efficient way, re-using parts of other solutions and maintaining the same common structure for all the solutions. We have called this structure a multi-tree. The accuracy of the first solution given by our algorithm is comparable to other decision tree systems, such as C5.0. Moreover, the multi-tree can be further populated in order to improve this accuracy. A new tree is not generated each time from scratch, but just some new branches are explored. This makes that once a first solution has been found the time which is required to produce the next n trees increases in a sublinear way.

This behaviour of the multi-tree algorithm is similar to an anytime algorithm, which increases the quality of the solution with increasing time. In [Jensen and Schlimmer, 1996], the authors present a technique that constructs a single decision tree in an anytime way.

We have studied several strategies to populate the multi-tree, and then to extract just one hypothesis, or a set of hypotheses. The extraction of a single hypothesis is based on Occam's razor: "simple solutions are better". Experimental results have shown that the Occam solution generally improves in accuracy and comprehensibility with respect to classical decision tree learning. On the other hand, we have also developed a method to extract a set of hypotheses from the multi-tree that will be accurate and diverse among them.

A decision multi-tree structure is similar to an option tree [Buntine, 1990, 1993; Kohavi and Kunz, 1997]. The main difference between these methods is the construction strategy. In the multi-tree method, we store all the rejected splits as suspended nodes, and later we continue the multi-tree construction selecting one of these suspended nodes according to a criterion. Option trees are constructed as usual decision trees, but when at a level the difference between the best test and the other alternatives (option factor) is small, these are also explored as optional tests (OR-nodes).

#	1	50	150	150	200
1	77,94	77,81	77,68	77,55	77,55
2	93,81	94,26	93,99	94,01	93,71
3	92,43	92,61	92,48	92,46	92,55
4	88,61	88,98	88,92	88,99	89,01
5	99,61	99,64	99,64	99,65	99,65
6	48,29	48,37	48,37	48,33	48,29
7	91,47	92,08	92,31	92,44	92,44
8	38,23	39,85	40,23	39,54	39,38
9	66,33	66,37	66,47	66,53	66,5
10	74,06	73,06	73,06	73,06	73,06
11	52,08	51,42	51,32	51,24	51,24
12	75,8	76,07	76,13	76,27	76,47
13	62,72	62,81	62,64	62,61	62,58
14	78,53	78,75	78,53	78,5	78,5
15	94,74	94,56	94,65	94,65	94,74
16	94,13	94,53	94,13	94,8	95
17	64,44	64,26	64,21	64,21	64,09
18	95,22	97,05	98,31	98,73	99,11
19	70,97	71,32	71,22	71	70,83
20	97,47	97,67	97,67	97,67	97,67
21	91,95	92,57	92,43	92,38	92,29
22	69,63	69,25	69,51	69,43	69,53
23	62,25	66,5	67	67	67,37
24	95,91	95,95	95,98	95,97	96
25	60,6	61,27	61,27	61,27	61,4
26	99,23	99,24	99,31	99,33	99,34
27	77,16	77,57	77,74	77,99	78,19
28	93	93,82	93,76	94,18	94,47
GeoMean	76,73	77,17	77,22	77,21	77,25

Table 9.4: Accuracy of the Occam solution depending on the size of the multi-tree.

#	1	50	100	150	200
1	0,824	0,822	0,822	0,821	0,821
2	0,979	0,979	0,979	0,979	0,978
3	0,968	0,968	0,968	0,968	0,968
4	0,953	0,947	0,947	0,948	0,948
5	1,000	1,000	1,000	1,000	1,000
6	0,657	0,657	0,657	0,657	0,657
7	0,991	0,991	0,992	0,992	0,992
8	0,622	0,633	0,639	0,639	0,635
9	0,641	0,639	0,640	0,640	0,641
10	0,900	0,872	0,872	0,872	0,872
11	0,667	0,662	0,660	0,661	0,661
12	0,718	0,715	0,721	0,723	0,725
13	0,724	0,724	0,724	0,724	0,724
14	0,823	0,825	0,824	0,824	0,823
15	0,987	0,986	0,986	0,986	0,986
16	0,987	0,987	0,987	0,987	0,987
17	0,667	0,666	0,666	0,666	0,666
18	0,987	0,993	0,997	0,998	0,999
19	0,710	0,714	0,713	0,708	0,705
20	0,991	0,990	0,990	0,990	0,990
21	0,970	0,970	0,969	0,967	0,969
22	0,761	0,759	0,760	0,760	0,760
23	0,559	0,581	0,582	0,585	0,600
24	0,997	0,997	0,997	0,997	0,997
25	0,763	0,763	0,761	0,760	0,760
26	0,998	0,998	0,999	0,999	0,999
27	0,873	0,873	0,874	0,875	0,876
28	0,976	0,978	0,978	0,980	0,982
GeoMean	0,833	0,833	0,834	0,834	0,835

Table 9.5: AUC of the Occam solution depending on the size of the multi-tree.

#	1	50	100	150	200
1	138,97	138,48	138,19	138,05	138,01
2	43,19	40,05	39,23	38,82	38,30
3	31,68	31,68	31,68	31,65	31,63
4	282,26	277,83	276,54	275,70	275,51
5	47,90	47,62	47,62	47,45	47,24
6	929,55	925,28	922,05	919,68	917,22
7	58,32	53,68	53,44	53,26	53,15
8	41,26	37,44	36,84	36,62	36,52
9	106,23	105,31	105,20	105,16	105,15
10	43,95	42,90	42,90	42,90	42,90
11	776,29	772,66	767,90	766,66	764,91
12	80,66	78,00	77,31	76,84	76,72
13	258,30	254,24	252,21	250,26	248,56
14	146,07	143,33	140,34	139,83	139,43
15	48,50	44,36	43,88	43,78	43,52
16	11,24	10,86	10,66	10,35	10,16
17	102,23	101,66	100,80	100,54	100,37
18	86,84	80,36	68,01	60,74	53,76
19	278,51	277,15	276,19	275,17	274,65
20	37,11	36,87	36,87	36,87	36,87
21	19,10	18,58	18,37	18,02	17,76
22	197,83	197,82	197,28	196,05	195,66
23	46,61	44,47	43,96	43,84	43,78
24	102,74	102,48	102,47	102,45	102,45
25	63,71	62,55	62,11	61,92	61,81
26	70,39	68,77	64,87	63,42	62,83
27	338,62	337,08	336,42	335,56	334,18
28	14,48	13,56	13,15	12,73	12,45
GeoMean	85,54	83,14	81,87	81,03	80,32

Table 9.6: Number of rules of the hypotheses depending on the size of the multi-tree.

The creation of optional nodes is limited at the upper levels to restrict the size of the option tree. Consequently, the size of the tree and the learning time are very difficult to be tuned depending on the problem. In [Kohavi and Kunz, 1997] Kohavi and Kunz discussed about this limitation: "We ran the option algorithm varying the option factor from 0.1 to 0.4, and imposed and arbitrary limit of five options per node to control the tree size. Using an option factor of 0.5 increased the run-times dramatically so we stopped at that point [....] Using 0.4 as the option factor produce huge trees; Soybean was the extreme case, which increased from 68 nodes to 203,577, a factor of almost 3000". Therefore, while options trees are constructed by a breadth search, a multi-tree is generated by a beam search, i.e. we store the discarded alternatives in a list of suspended nodes, and later, if we want to populate further the multi-tree, we select one node from the list according to one selection criterion, and at this point the search is continued. Moreover, the structure used in option trees is not employed to extract one or more solutions, it is just considered as one model. This model labels examples by combining the predictions of the options tests at the AND-nodes where more than one option has been considered.

Another relation of our multi-tree method can be established to some rule learning method. The beam search employed for the generation of the multi-tree structure is related to the rule learning system CN2 [Clark and Niblett, 1989]. This method performs a search in which the algorithm maintains a list of k candidates at each selection, instead of considering just a single option. At each search step, the algorithm generates descendants from these k candidates, and the resulting set is again reduced to the k most promising members. The main difference between the two learning algorithms is that CN2 is a rule learning method that learns rule by rule, while decision tree learning generates all the rules while the tree is populated. Moreover, operationally CN2 is not anytime, since the same value of k for different problems can give very different running times.

Summing up, we have presented an anytime algorithim for constructing decision trees. Accuracy and AUC are slightly improved, but not as much as it would be desired. Nonetheless, accuracy and AUC could even be improved if we combine the predictions of the models in the multi-tree. The next chapter introduces some developments about the use of a multi-tree structure as an ensemble method.

Chapter 10

From Shared Ensembles to Comprehensible Models

In Chapter 6, we have described ensemble methods as a popular technique to generate a composite classifier by combining a set (or ensemble) of classifiers, which usually increases the accuracy of any single component of the ensemble. We have also discussed some of the main drawbacks of ensemble methods, such as the loss of comprehensibility, or the high level of computational resources required by these methods.

In this chapter we study the combination of the hypotheses in the multi-tree. In this way the combination of these hypotheses can be considered as an ensemble method. In this case, this ensemble method has desirable properties, mainly due to the share of common parts of the hypotheses in the multi-tree. We also study some fusion techniques in order to take the maximum profit of the combined hypothesis. A description of the application of a multi-tree as an ensemble method has appeared in [Estruch *et al.*, 2002b,c, 2003].

In order to improve the comprehensibility of the combined model, but without degrading the accuracy or AUC, we define a technique that permits the extraction of the hypothesis from the ensemble that is most similar to their combination. A description of this method has been published in [Ferri *et al.*, 2002b].

10.1 Introduction

In Chapter 6, we have detailed ensemble methods as a useful technique to produce an accurate model by the combination of a set of models.

Ensemble methods require the generation, storage and application of a set of models in order to predict future cases. This represents an important consumption of resources, in both scenarios: learning process and predicting new cases.

Another, but not less important, drawback of ensemble methods, is the loss of comprehensibility that represents the combination of various models. A composite model becomes a black box, whose behaviour is difficult to be understood.

These important hindrances frequently limit the application of ensemble methods to real problems, such as medical decision making support, where comprehensibility is crucial. In this chapter we present two methods based on the multi-tree structure that partially overcome these limitations.

First, we define an ensemble method that combines the individual components of the multi-tree. This ensemble method takes profit of the internal structure of the multi-tree by sharing common parts of the components of the ensemble. In this way the quantity of resources required for the learning and the application of the ensemble is reduced. We call this new ensemble method: *shared ensemble*.

Secondly, we address the problem of loss of comprehensibility of ensemble methods. We introduce a technique that permits the generation of a single comprehensible model that is semantically similar to the shared ensemble. We call this single model *archetype*. Due to this similarity, the accuracy and AUC of the archetype model is improved with respect to the single models obtained with classical decision tree learning, or even the models selected in the multi-tree by other selection methods, such as Occam.

10.2 Shared Ensembles

Section 9.2 has introduced the multi-tree structure as a method to generate a set of different models, in this case decision trees. These decision trees have the peculiarity that some of their parts are shared. Until now, we have employed this structure just to select one model (or a set of models) according to a selection criterion (see Sections 9.2.2). However, since there is a set of models in the multi-tree, it could also be possible to combine them in order to predict new cases with more accuracy. This is the basic idea of ensemble methods (see Chapter 6).

The combination of the single components of a multi-tree presents singular properties with respect to classical ensemble methods. The main difference is the sharing of common parts of these components in the multi-tree. Due to this fact, we name this set of models a *shared ensemble*.

The sharing of some parts of the single models has several advantages with respect to other methods. Obviously, the main advantage is the better dealing of resources. A multi-tree is constructed faster than an ensemble of completely independent models, and it requires less memory for constructing and storing the components. Moreover, the use of resources for the prediction of new events, or throughput, also gets better.

Nevertheless, this sharing could also be a problem, since the quality of an ensemble method is based on the diversity of its components. If the models of the multi-tree share most of their parts, they could be too similar, degrading the accuracy of the ensemble method. We will see experimentally that the criterion that we employ to populate the tree represents a trade-off between sharing of components and the accuracy of the ensemble method.

Fusion Methods and Shared Ensembles

In Section 6.3 we have introduced well-known methods to compute the final class of an example given a set of classifiers. Usually, these methods are defined over classifiers that assign a probability to each prediction. Let us denote by $p_k(c_j|x)$ an estimate of the posterior probability that classifier k assigns class c_j for example x. In shared ensembles, $p_k(c_j|x)$ depends on the leaf where each x falls. More precisely, these probabilities are computed using the *proportion* of training examples of each class that have fallen into each rule during training. The reliability of each rule usually depends on the *cardinality* of the rule.

Example 14 *Consider three classes a, b and c and two nodes n_1 and n_2 with the following train distribution:*

$$n_1 = \{40, 10, 30\}$$
$$n_2 = \{0, 12, 6\}$$

If we convert these absolute values into relative values (similar to the $p_k(c_j|x)$), we would have:

$$r_1 = \{0.5, 0.125, 0.375\}$$
$$r_2 = \{0, 0.667, 0.333\}$$

According to the relative values, the prediction of class a by n_1 seems less probable than the prediction of class b by n_2. However, node n_1 has 40 examples supporting its prediction whereas n_2 has only 12 examples.

Consequently, in order to combine different predictions we must consider the class distribution in each rule as well as its cardinality; as a result of this rationale we will use the absolute values.

Let us define a *class vector* $v_{k,j}(x)$ as the vector of training cases that fall in each rule k for each class j.

Note that in a multi-tree the number of hypotheses increases exponentially with respect to the number of nodes explored. Since it could be prohibitive to combine all these models at the top, the combination is performed inside the multi-tree structure by a majority vote. For leaf nodes, the values would be the training cases of each class that have fallen into the leaf. To propagate these vectors upwards to internal nodes, we must clarify how to propagate through AND and OR nodes. This is done for each new unlabelled example we want to make a prediction for. For the OR-nodes, the answer is clear: an example can only fall through one of its children. Hence, the vector would be the vector of the child where the example falls. AND-nodes, however, must do a fusion whenever different alternate vectors occur. This is an important difference in shared ensembles: fusion points are distributed all over the multi-tree structure.

In Section 6.3, we have presented several fusion strategies that convert the class vectors computed by each classifier of the ensemble into one combined vector. In classical ensemble methods the fusion is performed with respect to the final vector of probabilities of each component of the ensemble. However, in a multi-tree the fusion is performed internally. Concretely, in each AND-node the partial results of its children OR-node are joined. Therefore, it is then at each AND-node where the specified fusion strategy is applied.

In addition, it is also possible to apply some transformations/modifications to the original vectors:

- **good loser** : $v'_{k,j}(x) = \sum_j v_{k,j}(x)$ if $j = \texttt{argmax}(v_{k,j}(x))$ and 0 otherwise
- **bad loser:** $v'_{k,j}(x) = v_{k,j}(x)$ if $j = \texttt{argmax}(v_{k,j}(x))$ and 0 otherwise.
- **majority:** $v'_{k,j}(x) = 1$ if $j= \texttt{argmax}(v_{k,j}(x))$ and 0 otherwise.
- **difference**: $v'_{k,j}(x) = v_{k,j}(x) - \sum_{i \neq j} v_{k,j}(x)$
- **squared**: $v'_{k,j}(x) = [v_{k,j}(x)]^2$

In the literature, only the original vectors have been considered. We have introduced these transformations to see whether they can be used to increase accuracy or AUC.

Example 15 *Given a set of vectors (*original*), the following table shows the results of applying the transformations to two vectors.*

Original	Good loser	Bad loser	Majority	Difference	Squared
{ 40, 10, 30 }	{ 80, 0, 0 }	{ 40, 0, 0}	{ 1, 0, 0 }	{ 0, -60, -20 }	{ 1600, 100, 900 }
{ 7, 2, 10 }	{ 0, 0, 19 }	{ 0, 0, 10 }	{ 0, 0, 1 }	{ -5, -15, 1 }	{ 49, 4, 100 }
{ 0, 10, 1 }	{ 0, 11, 0 }	{ 0, 10, 0 }	{ 0, 1, 0 }	{ -11, 9, -9 }	{ 0, 100, 1 }
{ 5, 6, 3 }	{ 0, 14, 0}	{ 0, 6, 0 }	{ 0, 1, 0 }	{ -4, -2, -8 }	{ 25, 36, 9 }

Obviously, each transformation and fusion strategies will have different effect.

10.3 Experiments

In this section, we study experimentally the behaviour of the shared ensemble depending on some factors. First we evaluate the application of some fusion methods. Then, we analyse the accuracy of the shared ensemble combination depending on the use of some modifications over the vectors. We also investigate which suspended OR-node selection criterion produces better ensembles. In this case, since a random selection seems to behave well, we introduce a method that highly improves the performance, based on the rejection of some nodes in the construction process. Finally, we analyse the improvement in terms of accuracy of the ensemble method, and we compare it with other well-known ensemble methods.

10.3.1 Fusion Strategies Evaluation

Here, we present an experimental comparison of the fusion methods presented in the previous section. With this aim, we have implemented the shared ensembles in the SMILES system (see Appendix C). The experiments have been obtained by constructing a multi-tree where 100 suspended OR-nodes have been explored by selecting them randomly.

Table 10.1 contains an accuracy comparison of these fusion methods for the combination, while Table 10.2 reveals the results in AUC value. The results indicate that arithmetic, maximum and sum are the best methods. The behaviour of the three methods are quite similar; concretely, in terms of accuracy, the maximum method obtains the highest accuracy, while in terms of the AUC measure, the best one is arithmetic. A possible reason of the bad results of the geometric, minimum and product fusion methods could be because the predictions are drastically affected when any of the vectors that are joined contains a 0 in one of their components. The application of a smoothing technique, such as Laplace correction could probably improve the results of such fusion methods.

As a result of these experiments, in the following, we will use maximum as fusion method.

Vector modification methods

Table 10.3 illustrates the results for accuracy, and Table 10.4 the results for AUC, when using the *original* vector and the *bad loser*, *difference*, *good loser*, *majority* and *squared* transformations. These methods have been introduced in the previous section. The modifications are applied in all the leaves once the multi-tree has been constructed.

The results have been obtained by constructing a multi-tree where 100 suspended OR-nodes have been explored by selecting them randomly. According to these exper-

#	Arithmetic	Geometric	Maximum	Minimum	Product	Sum
1	80.39	76.97	82.52	76.97	76.97	80.37
2	91.46	83.37	91.31	83.37	83.37	91.52
3	99.52	99.44	99.37	99.44	99.44	99.51
4	95.31	90.67	93.72	90.67	90.67	95.36
5	37.00	36.54	36.62	36.54	36.54	35.92
6	81.56	77.06	76.75	77.06	77.06	76.94
7	95.74	93.93	95.74	93.93	93.93	95.81
8	94.40	93.40	95.47	93.40	93.40	94.27
9	67.85	63.88	67.71	63.88	63.88	68.32
10	99.71	84.58	100.00	84.58	84.58	99.71
11	70.78	72.45	76.52	72.45	72.45	71.13
12	97.98	97.56	97.87	97.56	97.56	98.00
13	94.67	93.00	92.90	93.00	93.00	94.14
14	73.55	69.07	73.50	69.07	69.07	73.78
15	94.99	93.70	94.93	93.70	93.70	95.16
16	94.04	91.70	93.25	91.70	91.70	93.39
17	49.56	47.84	49.89	47.84	47.84	49.63
18	66.93	65.63	68.67	65.63	65.63	66.80
19	53.87	50.68	54.92	50.68	50.68	53.79
20	80.00	75.13	81.73	75.13	75.13	80.27
21	66.97	63.00	66.64	63.00	63.00	67.33
22	83.19	76.28	83.19	76.28	76.28	83.94
23	61.38	62.25	62.50	62.25	62.25	61.50
24	99.33	99.19	99.30	99.19	99.19	99.34
25	96.52	95.83	96.32	95.83	95.83	96.55
26	61.87	61.60	63.00	61.60	61.60	61.47
27	81.23	78.48	82.16	78.48	78.48	81.29
28	95.00	91.59	93.00	91.59	91.59	94.00
Gmean	78.74	75.96	78.99	75.96	75.96	78.52

Table 10.1: Accuracy comparison of some fusion methods in the multi-tree

#	Arithmetic	Geometric	Maximum	Minimum	Product	Sum
1	0,833	0,795	0,798	0,810	0,810	0,815
2	0,987	0,972	0,985	0,977	0,977	0,987
3	0,982	0,961	0,978	0,967	0,967	0,981
4	0,952	0,891	0,939	0,888	0,888	0,949
5	1,000	0,999	0,999	0,999	0,999	0,999
6	0,676	0,651	0,681	0,653	0,653	0,673
7	0,998	0,989	0,998	0,989	0,989	0,999
8	0,629	0,559	0,640	0,582	0,582	0,628
9	0,660	0,631	0,665	0,648	0,648	0,643
10	0,964	0,929	0,960	0,933	0,933	0,959
11	0,685	0,651	0,690	0,654	0,654	0,688
12	0,802	0,743	0,821	0,753	0,753	0,809
13	0,787	0,710	0,782	0,718	0,718	0,803
14	0,859	0,804	0,858	0,813	0,813	0,870
15	0,989	0,979	0,988	0,982	0,982	0,989
16	0,994	0,989	0,995	0,988	0,988	0,994
17	0,715	0,669	0,716	0,668	0,668	0,714
18	1,000	0,931	1,000	0,928	0,928	1,000
19	0,779	0,721	0,769	0,727	0,727	0,784
20	0,992	0,990	0,991	0,989	0,989	0,991
21	0,989	0,975	0,992	0,976	0,976	0,993
22	0,791	0,757	0,794	0,760	0,760	0,784
23	0,413	0,531	0,357	0,525	0,525	0,395
24	0,998	0,996	0,998	0,996	0,996	0,998
25	0,789	0,767	0,791	0,769	0,769	0,795
26	0,998	0,996	0,998	0,997	0,997	0,998
27	0,914	0,867	0,915	0,876	0,876	0,902
28	0,993	0,971	0,992	0,973	0,973	0,993
Gmean	0,847	0,822	0,842	0,827	0,827	0,845

Table 10.2: AUC comparison of some fusion methods in the multi-tree

iments, all transformations get very similar results, except from *majority*. The poor results of the *majority* could be explained because this transformation is very drastic, and it simplifies too much the information at each leaf.

According to the previous results, we will use the combination *Maximum + Original* in the following experiments, because it gives the best results.

Suspended OR-node Selection Criterion

In this part we analyse the behaviour of the quality of the shared ensemble combination depending on some criteria to populate the multi-tree: *Bottom, Optimal, Random, Rival Ratio* and *Top Most* (see Section 9.2.1).

Table 10.5 shows the accuracy comparison among these methods for populating the multi-tree. Table 10.6 contains the results in AUC. There are many interesting things to be observed in these tables. According to the results, the best methods to build the shared ensemble for combination seem to be *Random* and *Top Most* in accuracy, and *Random* and *Rival Ratio* in AUC. The *Top Most* method has an important drawback: the decision trees do not share many components, so the construction and use of the shared ensemble is slower. This fact is detected if one observes the consumption of time. Table 10.7 exhibits the average learning time for each dataset depending on the method employed to populate the multi-tree. The difference in time between *Random* and *Top Most* is quite important. This is because the *Top Most* method shares fewer parts between trees and hence requires more memory and time.

Therefore, the *Random* criterion can be considered as an optimal trade-off between efficiency and accuracy/AUC when the multi-tree is constructed to combine its components.

Node forgetting

Despite the advantages of the multi-tree structure, many suspended nodes are never 'woken', occupying memory needlessly. Since we have seen that a good criterion is to select the nodes randomly, an additional criterion can be specified to forget some of the suspended nodes and, hence, to use less memory. Some criteria for forgetting nodes that we have considered are: to leave a constant minimum number of nodes, and to leave a logarithmic number of nodes. These criteria can be corrected by the depth of the nodes, because nodes on the top are more significant and usually give more different solutions.

Specifically, we have studied the following methods to restrict the nodes to be maintained in the list of the suspended nodes:

- **Constant:** We only store a constant minimum number of nodes.

#	Bad loser	Difference	Good loser	Majority	Original	Squared
1	82,48	82,48	82,48	69,71	82,52	82,48
2	94,91	94,91	94,91	86,52	94,93	94,91
3	93,07	93,07	93,07	83,50	93,25	93,07
4	91,20	91,20	91,20	80,40	91,31	91,20
5	99,38	99,38	99,38	98,89	99,37	99,38
6	49,91	49,82	49,88	48,13	49,89	49,91
7	93,83	93,83	93,83	77,47	93,72	93,83
8	37,54	33,77	36,62	36,77	36,62	37,54
9	67,70	67,70	67,93	69,27	68,67	67,70
10	76,75	76,75	76,75	58,88	76,75	76,75
11	54,10	53,77	54,21	51,13	54,92	54,10
12	81,53	81,53	81,53	79,13	81,73	81,53
13	67,00	67,03	67,00	65,69	66,64	67,00
14	83,36	83,75	83,36	60,25	83,19	83,36
15	95,67	95,67	95,67	90,07	95,74	95,67
16	95,33	95,33	95,33	90,27	95,47	95,33
17	67,62	67,62	67,62	51,91	67,71	67,62
18	99,78	99,78	99,78	79,56	100,00	99,78
19	75,68	75,68	75,68	82,70	76,52	75,68
20	97,85	97,85	97,85	91,89	97,87	97,85
21	93,33	93,33	93,33	88,14	92,90	93,33
22	73,99	73,99	73,99	65,21	73,50	73,99
23	63,25	63,00	63,25	63,88	62,50	63,25
24	96,33	96,33	96,33	94,50	96,32	96,33
25	62,47	63,93	62,47	61,93	63,00	62,47
26	99,27	99,27	99,27	98,44	99,30	99,27
27	82,01	82,01	82,01	68,55	82,16	82,01
28	93,06	93,06	93,06	78,94	93,00	93,06
Gmean	78,98	78,72	78,92	72,09	78,99	78,98

Table 10.3: Accuracy comparison of some vector modifications in the multi-tree.

#	Bad loser	Difference	Good loser	Majority	Original	Squared
1	0,794	0,794	0,794	0,616	0,798	0,801
2	0,985	0,985	0,985	0,973	0,985	0,984
3	0,981	0,981	0,981	0,953	0,978	0,981
4	0,939	0,939	0,939	0,892	0,939	0,940
5	1,000	1,000	1,000	0,998	0,999	0,999
6	0,683	0,684	0,682	0,637	0,681	0,683
7	0,998	0,998	0,998	0,986	0,998	0,998
8	0,607	0,575	0,601	0,574	0,640	0,627
9	0,662	0,661	0,662	0,614	0,665	0,660
10	0,948	0,945	0,947	0,899	0,960	0,957
11	0,685	0,687	0,683	0,634	0,690	0,684
12	0,808	0,808	0,808	0,706	0,821	0,809
13	0,793	0,792	0,793	0,734	0,782	0,789
14	0,863	0,864	0,863	0,768	0,858	0,861
15	0,989	0,989	0,989	0,970	0,988	0,988
16	0,994	0,994	0,994	0,982	0,995	0,994
17	0,707	0,707	0,707	0,631	0,716	0,707
18	1,000	1,000	1,000	0,954	1,000	1,000
19	0,764	0,764	0,764	0,725	0,769	0,764
20	0,992	0,992	0,992	0,978	0,991	0,992
21	0,992	0,992	0,992	0,981	0,992	0,992
22	0,795	0,795	0,795	0,716	0,794	0,797
23	0,404	0,392	0,413	0,496	0,357	0,361
24	0,998	0,998	0,998	0,996	0,998	0,998
25	0,787	0,789	0,783	0,781	0,791	0,786
26	0,998	0,998	0,998	0,993	0,998	0,998
27	0,913	0,913	0,913	0,795	0,915	0,918
28	0,992	0,992	0,992	0,974	0,992	0,993
Gmean	0,843	0,841	0,843	0,803	0,842	0,841

Table 10.4: AUC comparison of some vector modifications in the multi-tree.

#	Bottom	Optimal	Random	Rival Ratio	Top most
1	78,18	79,00	82,61	77,63	86,60
2	93,84	94,70	94,88	93,45	94,45
3	92,43	92,55	93,07	93,75	93,75
4	89,86	89,94	91,20	89,01	91,65
5	99,62	99,58	99,38	99,64	99,62
6	48,35	48,23	49,73	48,31	51,70
7	91,86	92,92	94,03	92,17	91,31
8	37,77	38,54	37,38	35,85	34,62
9	66,30	66,40	67,33	65,37	70,17
10	76,44	76,75	76,75	73,06	76,75
11	52,23	52,48	54,43	52,20	59,13
12	76,07	75,87	81,40	75,93	83,67
13	62,72	62,78	67,28	62,69	75,28
14	78,53	78,00	83,36	78,33	85,83
15	94,70	95,53	95,67	94,35	95,23
16	94,13	94,13	95,33	94,20	94,67
17	64,53	63,44	67,47	65,29	70,59
18	95,20	95,91	99,78	99,73	100,00
19	75,42	75,33	75,67	71,05	77,55
20	98,02	98,02	97,85	97,38	97,95
21	92,14	93,38	93,29	93,14	92,43
22	69,62	69,49	73,68	71,71	69,62
23	64,62	63,75	63,25	69,87	63,00
24	95,91	95,91	96,40	96,06	96,06
25	60,47	60,60	62,93	60,07	63,40
26	99,24	99,37	99,26	99,23	99,19
27	77,21	76,97	82,01	78,13	84,94
28	92,94	92,59	93,06	94,00	90,00
Geomean	77,15	77,27	78,98	77,16	79,77

Table 10.5: Accuracy of combination depending on the second tree criterion.

#	Bottom	Optimal	Random	Rival Ratio	Top most
1	0,821	0,814	0,797	0,813	0,820
2	0,979	0,986	0,986	0,980	0,974
3	0,968	0,968	0,981	0,972	0,984
4	0,946	0,938	0,939	0,954	0,952
5	1,000	1,000	1,000	1,000	1,000
6	0,657	0,658	0,680	0,671	0,625
7	0,991	0,993	0,998	0,993	0,995
8	0,652	0,643	0,631	0,642	0,622
9	0,640	0,638	0,660	0,650	0,654
10	0,952	0,960	0,960	0,960	0,872
11	0,667	0,667	0,693	0,673	0,678
12	0,719	0,732	0,809	0,774	0,746
13	0,724	0,730	0,794	0,741	0,775
14	0,823	0,827	0,863	0,838	0,834
15	0,987	0,988	0,989	0,988	0,976
16	0,987	0,990	0,994	0,993	0,988
17	0,667	0,666	0,710	0,717	0,711
18	0,990	0,994	1,000	1,000	1,000
19	0,778	0,766	0,764	0,771	0,662
20	0,992	0,991	0,992	0,991	0,990
21	0,971	0,982	0,992	0,981	0,983
22	0,761	0,761	0,794	0,796	0,761
23	0,500	0,545	0,367	0,515	0,547
24	0,997	0,997	0,998	0,998	0,998
25	0,762	0,765	0,791	0,770	0,759
26	0,998	0,999	0,998	0,998	0,999
27	0,872	0,870	0,913	0,887	0,875
28	0,977	0,991	0,993	0,989	0,973
Geomean	0,835	0,839	0,842	0,846	0,835

Table 10.6: AUC of combination depending on the second tree criterion.

#	Bottom	Optimal	Random	Rival Ratio	Top most
1	0,05	0,06	1,10	0,08	31,41
2	0,81	0,34	2,47	0,29	21,92
3	0,09	0,56	11,86	8,77	136,38
4	0,05	0,09	0,43	0,10	1,47
5	0,39	0,63	5,78	0,91	0,30
6	0,56	0,46	5,64	40,11	939,99
7	0,30	0,33	3,51	1,09	16,03
8	0,12	0,08	0,80	0,17	3,27
9	0,04	0,05	0,98	3,93	34,88
10	0,04	0,02	0,07	0,03	0,03
11	0,40	0,36	5,00	5,64	255,20
12	0,06	0,07	1,52	0,13	7,92
13	0,21	0,18	3,59	0,58	83,75
14	0,17	0,14	3,19	0,26	41,93
15	0,02	0,08	0,54	0,07	2,62
16	0,03	0,12	0,35	0,05	1,59
17	0,01	0,17	6,10	135,59	76,48
18	0,16	0,02	0,16	0,04	0,26
19	0,02	0,03	0,29	0,07	1,05
20	0,03	0,02	0,15	0,02	0,18
21	0,01	0,16	0,91	0,96	6,93
22	0,03	0,76	12,26	147,85	0,81
23	0,01	0,02	0,21	0,03	0,48
24	1,32	2,71	15,92	28,78	15,82
25	0,03	0,04	0,85	11,40	118,58
26	1,24	2,17	11,31	7,56	14,62
27	0,05	0,05	0,46	0,06	9,02
28	0,06	0,57	1,99	0,68	880,00
Geomean	0,09	0,14	1,43	0,68	8,55

Table 10.7: Average learning time (in seconds) depending on the second tree criterion.

- **Logarithmic:** A logarithmic number of nodes is preserved (a constant minimum is also applied).

- **Logarithmic + depth:** A logarithmic number of nodes is maintained but corrected by the depth of the nodes (a constant minimum is also applied). This correction tends to store more nodes at the top positions.

A comparison of these criteria according to the accuracy and AUC of the combination is presented in Table 10.8 and Table 10.9. The results of these tables indicate that the forgetting process does not produce significant modifications on the quality of the classifiers. This is due to the random strategy of selecting nodes. Consequently, we can conclude that, when using a random construction, forgetting methods save more memory and time, without any other bothering.
Table 10.10 contains the average learning time for each classifier and dataset, and the geometric mean of all the datasets. The experiments demonstrate the usefulness of node forgetting, because it reduces the learning time. The best one is the criterion based on the preserving of a logarithmic number of all possible nodes. The version corrected by the depth is a bit slower because it selects nodes on the top of the multi-tree. The method based on a constant number also improves in time, but the results in accuracy and AUC are a bit worse than the logarithmic method.

Regarding the improvement of memory, Table 10.11 shows the use of memory (in Kilobytes) required to learn the multi-trees for a selection of the datasets. We also include the percentage of memory required by the forgetting method with respect to the original method (no forgetting). For large and medium problems the use of forgetting reduces drastically the use of memory. However, for small problems the reduction is more limited. The forgetting method that requires less memory is "logarithmic".

According to these experiments, the best suspended node-forgetting criterion seems to be randomly leaving a logarithmic number of nodes, without taking depth into account.

Size of the Multi-tree

Finally, let us study the influence of the size of the multi-tree on the combination. Table 10.12 and Table 10.13 present the accuracy and AUC of the first solution and the combined solution using a multi-tree with different number of explored OR-nodes (50, 100, 150, and 200) always selected by the random method and applying forgetting (logarithmic criterion).

The results indicate that accuracy and AUC are increased as long as further OR-nodes are explored in the multi-tree. However, if we study the table in detail, we can observe that this is a general behaviour, but an excessive increase in explored

#	No forgetting	Const=5	Logarithmic	Log. + depth
1	82.61	81.40	82.24	85.11
2	94.88	95.16	94.93	94.81
3	93.07	93.36	93.41	93.80
4	91.20	91.20	91.55	89.20
5	99.38	99.39	99.37	99.34
6	49.73	49.30	49.51	52.65
7	94.03	94.53	94.42	94.39
8	37.38	37.00	38.46	35.15
9	67.33	68.13	68.00	69.30
10	76.75	76.75	76.44	75.94
11	54.43	54.09	54.26	55.58
12	81.40	81.53	80.67	81.67
13	67.28	66.00	67.42	67.06
14	83.36	83.36	83.19	83.67
15	95.67	95.86	95.91	95.79
16	95.33	95.07	95.00	95.13
17	67.47	67.94	67.79	68.85
18	99.78	99.78	99.89	97.56
19	75.67	75.67	76.38	71.50
20	97.85	97.85	97.75	97.67
21	93.29	93.29	93.57	92.29
22	73.68	72.95	73.26	75.43
23	63.25	62.00	61.88	62.13
24	96.40	96.31	96.61	96.43
25	62.93	62.40	62.87	63.40
26	99.26	99.28	99.33	99.24
27	82.01	82.22	81.78	83.96
28	93.06	92.88	93.76	92.76
Geomean	78.98	78.79	79.03	79.02

Table 10.8: Accuracy of the combination in the multi-tree depending on the forgetting method.

#	No forgetting	Const=5	Logarithmic	Log. + depth
1	0.797	0.800	0.810	0.778
2	0.986	0.985	0.984	0.987
3	0.981	0.979	0.980	0.983
4	0.939	0.939	0.939	0.952
5	1.000	0.999	0.999	0.999
6	0.680	0.673	0.676	0.713
7	0.998	0.998	0.998	0.998
8	0.631	0.641	0.653	0.633
9	0.660	0.663	0.665	0.677
10	0.960	0.960	0.936	0.925
11	0.693	0.682	0.686	0.701
12	0.809	0.813	0.805	0.825
13	0.794	0.778	0.788	0.802
14	0.863	0.861	0.859	0.861
15	0.989	0.989	0.988	0.990
16	0.994	0.994	0.996	0.995
17	0.710	0.704	0.708	0.736
18	1.000	1.000	1.000	0.994
19	0.764	0.764	0.760	0.708
20	0.992	0.992	0.992	0.991
21	0.992	0.994	0.993	0.995
22	0.794	0.790	0.789	0.814
23	0.367	0.354	0.375	0.416
24	0.998	0.998	0.998	0.998
25	0.791	0.783	0.790	0.807
26	0.998	0.999	0.998	0.999
27	0.913	0.916	0.912	0.929
28	0.993	0.994	0.994	0.995
Geomean	0.842	0.840	0.843	0.849

Table 10.9: AUC of the combination in the multi-tree depending on the forgetting method.

#	No forgetting	Const=5	Logarithmic	Log + depth
1	1.104	0.596	0.619	1.709
2	2.466	1.623	1.658	2.818
3	11.858	12.472	11.997	15.196
4	0.427	0.428	0.445	0.124
5	5.783	5.682	4.768	9.973
6	5.644	3.738	2.478	12.493
7	3.508	3.359	3.102	4.018
8	0.802	0.506	0.593	1.057
9	0.978	0.810	0.707	1.551
10	0.071	0.074	0.020	0.012
11	4.997	2.658	2.645	5.543
12	1.522	1.385	1.359	1.875
13	3.586	2.339	2.448	3.698
14	3.192	2.329	2.285	3.580
15	0.543	0.508	0.469	0.941
16	0.346	0.302	0.274	0.526
17	6.096	6.663	4.618	10.464
18	0.164	0.172	0.144	0.043
19	0.288	0.303	0.286	0.082
20	0.152	0.160	0.121	0.030
21	0.912	0.903	0.744	1.267
22	12.264	18.813	9.963	11.086
23	0.210	0.179	0.157	0.080
24	15.921	17.337	16.171	17.444
25	0.849	0.626	0.569	1.278
26	11.309	12.142	9.786	15.357
27	0.465	0.444	0.370	0.548
28	1.986	2.055	1.695	2.051
Geomean	1.426	1.263	1.069	1.330

Table 10.10: Average learning time for the combined solution in the multi-tree depending on the forgetting method.

#	Original	Const=5	%	Logarithmic	%	Log + depth	%
4	5700.00	5700.00	100.00%	1628.00	28.56%	4608.00	80.84%
5	49512.00	10892.00	22.0%0	6388.00	12.90%	8328.00	16.82%
19	4880.00	4820.00	98.77%	1484.00	30.41%	3472.00	71.15%
28	18800.00	3268.00	17.38%	2272.00	12.09%	2608.00	13.87%

Table 10.11: Average memory (in Kbytes) for the combined solution in the multi-tree depending on the forgetting method.

OR-nodes can even degrade the results of some datasets. Therefore the size of the multi-tree to obtain optimal results depends on the particular features (size, number and type of attributes, number of classes) of each dataset. It could be interesting to study a method to estimate from the features of a dataset the approximate size of the multi-tree where the combination obtains the peak in accuracy.

Comparison with other ensemble methods

Let us compare the behaviour of the shared ensemble technique with other popular ensemble methods: *Bagging* and *Boosting* (Adaboost). We have employed the Weka (version 3.2.3)[1] implementation of these two ensemble methods. These ensemble methods use *J*48 as base classifier (the Weka version of C4.5), and we have used the default settings.
Pruning is only enabled for *Boosting*, since this method requires pruning to get good results. The experiments were executed on a Pentium III 800 Mhz with 175 MB of memory running Linux version 2.4.2. In this section, we do not study the effect of the ensemble methods in the AUC measure because Weka does not compute this value.

Table 10.14 includes the results in accuracy depending on the number of iterations: 1, 40, 80. The best results are obtained by the *Boosting* method. Note that the results for one iteration (i.e. the original C4.5 algorithm) are different, this is because of implementation details between SMILES and Weka, and because *Boosting* is applied with pruning, while the other two methods are not.

Table 10.15 shows the average training time required by the ensemble methods for each dataset, and the geometric mean for all the datasets depending on the number of iterations (1, 40, 80). The results indicate that as expected the training time is increased as long as the number of iterations is raised. The shared ensemble method requires much less time than the other two methods and this difference becomes greater as further iterations are executed.

With the aim of performing a better analysis of the accuracy behaviour of the three ensemble methods, we also include Figure 10.1. In this figure we exhibit the mean accuracy obtained by the three methods depending on the number of iterations.

[1] `http://www.cs.waikato.ac.nz/~ml/weka/`

#	1	50	100	150	200
1	77,94	80,61	82,24	83,11	83,61
2	93,81	94,83	94,93	94,96	94,91
3	92,43	93,32	93,41	93,07	92,87
4	88,61	90,98	91,55	91,45	91,52
5	99,61	99,44	99,37	99,32	99,30
6	48,29	48,93	49,51	50,08	49,63
7	91,47	93,97	94,42	94,36	94,11
8	38,23	36,15	38,46	36,54	37,85
9	66,33	67,43	68,00	68,50	68,27
10	74,06	76,44	76,44	76,44	76,44
11	52,08	53,84	54,26	54,84	54,67
12	75,80	80,07	80,67	82,00	82,07
13	62,72	66,28	67,42	66,61	66,83
14	78,53	83,03	83,19	83,25	83,83
15	94,74	95,60	95,91	95,84	95,93
16	94,13	94,67	95,00	95,00	95,27
17	64,44	66,88	67,79	68,12	67,65
18	95,22	99,44	99,89	99,93	100,00
19	70,97	74,03	76,38	77,97	79,13
20	97,47	97,91	97,75	97,69	97,69
21	91,95	93,29	93,57	93,19	92,86
22	69,63	72,82	73,26	74,11	73,45
23	62,25	61,25	61,88	63,25	63,50
24	95,91	96,23	96,61	96,23	96,30
25	60,60	61,40	62,87	62,87	63,00
26	99,23	99,30	99,33	99,29	99,22
27	77,16	80,47	81,78	82,37	83,06
28	93,00	93,24	93,76	92,59	92,41
Geomean	76,73	78,27	79,03	79,12	79,25

Table 10.12: Accuracy of the combination in the multi-tree depending on the size.

#	1	50	100	150	200
1	0.824	0.805	0.803	0.804	0.798
2	0.953	0.985	0.983	0.986	0.986
3	1.000	0.978	0.979	0.982	0.981
4	0.991	0.939	0.937	0.935	0.936
5	0.622	1.000	1.000	1.000	0.999
6	0.900	0.668	0.678	0.682	0.683
7	0.987	0.997	0.998	0.998	0.999
8	0.987	0.655	0.643	0.645	0.653
9	0.667	0.659	0.670	0.669	0.672
10	0.987	0.936	0.936	0.936	0.936
11	0.710	0.680	0.687	0.694	0.688
12	0.991	0.804	0.813	0.808	0.828
13	0.970	0.780	0.789	0.792	0.794
14	0.761	0.854	0.857	0.863	0.862
15	0.979	0.988	0.989	0.988	0.989
16	0.968	0.993	0.994	0.996	0.996
17	0.657	0.695	0.711	0.709	0.709
18	0.641	1.000	1.000	1.000	1.000
19	0.667	0.742	0.759	0.775	0.785
20	0.718	0.992	0.992	0.991	0.991
21	0.724	0.991	0.989	0.994	0.992
22	0.823	0.783	0.788	0.794	0.796
23	0.559	0.411	0.387	0.356	0.384
24	0.998	0.998	0.998	0.998	0.998
25	0.997	0.780	0.783	0.794	0.794
26	0.763	0.998	0.999	0.998	0.998
27	0.873	0.904	0.911	0.913	0.921
28	0.976	0.994	0.992	0.994	0.993
Geomean	0.833	0.841	0.843	0.843	0.846

Table 10.13: AUC of the combination in the multi-tree depending on the size.

	1			40			80		
#	Bag.	Boos	Multi.	Bag.	Boos	Multi.	Bag.	Boos	Multi.
1	79.40	77.70	77.94	82.85	76.62	80.11	82.92	76.16	81.90
2	94.12	94.48	93.81	96.11	96.53	94.81	96.27	96.64	95.14
3	94.22	94.32	92.43	95.91	96.84	93.13	95.98	97.10	93.25
4	94.37	92.70	88.61	94.17	96.88	90.86	94.27	97.05	91.33
5	99.40	99.40	99.61	99.40	99.62	99.45	99.41	99.64	99.37
6	49.98	51.98	48.29	52.80	50.38	48.97	52.66	50.38	49.45
7	93.80	93.88	91.47	96.89	96.48	94.31	97.08	96.51	94.28
8	67.32	67.86	38.23	66.21	67.61	36.46	66.23	67.61	37.15
9	70.20	70.75	66.33	71.67	71.17	66.90	71.73	71.17	68.47
10	72.50	74.19	74.06	54.25	65.44	76.44	56.13	65.88	76.44
11	47.26	54.78	52.08	46.45	54.78	53.21	46.58	54.78	54.02
12	78.98	79.22	75.80	82.38	84.52	79.73	82.57	84.46	81.60
13	65.82	61.48	62.72	65.77	61.48	65.64	65.77	61.48	66.58
14	83.15	81.52	78.53	83.12	81.52	82.56	83.12	81.52	83.28
15	95.47	96.46	94.74	96.55	95.08	95.40	96.64	95.08	95.77
16	94.60	94.67	94.13	94.33	94.47	94.27	94.33	94.60	95.00
17	65.59	65.41	64.44	73.03	70.60	66.47	73.32	70.63	68.21
18	95.11	98.35	95.22	100.00	99.46	99.35	100.00	99.46	99.75
19	62.66	64.05	70.97	66.91	80.44	73.43	66.99	81.70	75.75
20	98.67	98.92	97.47	98.90	97.92	98.00	98.88	97.92	97.80
21	92.95	93.05	91.95	94.56	95.13	93.48	94.90	95.27	93.24
22	73.40	73.74	69.63	76.16	73.39	71.51	76.47	73.59	72.97
23	59.89	70.44	62.25	64.44	59.00	62.75	65.11	59.00	63.25
24	96.90	96.90	95.91	97.69	98.51	96.43	97.64	98.57	96.40
25	56.46	55.73	60.60	61.50	64.81	61.93	60.71	64.81	62.33
26	99.72	99.68	99.23	99.65	99.70	99.32	99.65	99.72	99.28
27	79.16	80.31	77.16	83.82	82.70	80.14	83.82	82.72	81.36
28	93.49	93.43	93.00	95.79	96.46	93.71	95.90	96.62	92.18
GMean	78.75	79.72	76.73	80.00	80.79	78.17	80.16	80.88	78.85

Table 10.14: Accuracy comparison among ensemble methods.

	1			40			80		
#	Bag.	Boos	Multi.	Bag.	Boos	Multi.	Bag.	Boos	Multi.
1	0.08	0.08	0.04	2.29	4.50	0.11	3.82	9.66	0.18
2	0.06	0.06	0.08	2.32	4.48	0.29	4.06	9.23	0.45
3	0.29	0.29	0.45	8.73	14.26	1.88	16.96	29.04	3.15
4	0.07	0.07	0.03	1.65	4.75	0.09	3.01	8.91	0.14
5	0.53	0.53	0.31	12.44	42.53	0.84	23.94	80.70	1.27
6	0.40	0.40	0.27	8.99	5.65	0.51	17.26	5.49	0.85
7	0.04	0.04	0.32	1.28	2.37	0.47	2.55	4.63	0.77
8	0.01	0.01	0.02	0.22	0.02	0.09	0.41	0.03	0.15
9	0.01	0.01	0.03	0.58	0.06	0.12	1.16	0.07	0.20
10	0.01	0.01	0.00	0.35	0.29	0.01	0.74	0.38	0.01
11	0.18	0.18	0.31	10.43	0.40	0.51	22.73	0.38	0.73
12	0.03	0.03	0.06	0.75	1.33	0.19	1.53	2.64	0.32
13	0.03	0.03	0.19	1.10	0.07	0.39	1.99	0.07	0.60
14	0.04	0.04	0.14	2.06	0.11	0.37	4.06	0.11	0.59
15	0.03	0.03	0.02	0.78	0.71	0.08	1.57	0.74	0.12
16	0.01	0.01	0.01	0.18	0.21	0.04	0.37	0.26	0.07
17	0.06	0.06	0.08	1.60	0.98	0.61	3.23	1.11	1.34
18	0.02	0.02	0.01	0.47	0.13	0.03	0.99	0.14	0.04
19	0.04	0.04	0.02	0.91	1.75	0.06	1.95	3.37	0.09
20	0.01	0.01	0.01	0.29	0.46	0.02	0.59	0.48	0.04
21	0.02	0.02	0.02	0.49	0.66	0.12	1.00	1.22	0.20
22	0.13	0.13	0.23	5.07	8.11	1.35	10.36	14.97	3.02
23	0.01	0.01	0.00	0.19	0.08	0.03	0.42	0.09	0.04
24	1.17	1.17	1.04	33.56	57.33	2.50	68.30	116.43	4.28
25	0.02	0.02	0.02	0.55	0.22	0.09	1.13	0.24	0.17
26	0.69	0.69	3.91	18.95	100.32	2.39	38.31	210.50	3.51
27	0.06	0.06	0.04	1.43	3.09	0.07	3.23	6.31	0.11
28	0.03	0.03	0.05	0.88	1.15	0.25	1.78	2.15	0.42
GMean	0.05	0.05	0.06	1.47	1.13	0.19	2.94	1.67	0.30

Table 10.15: Time comparison among ensemble methods.

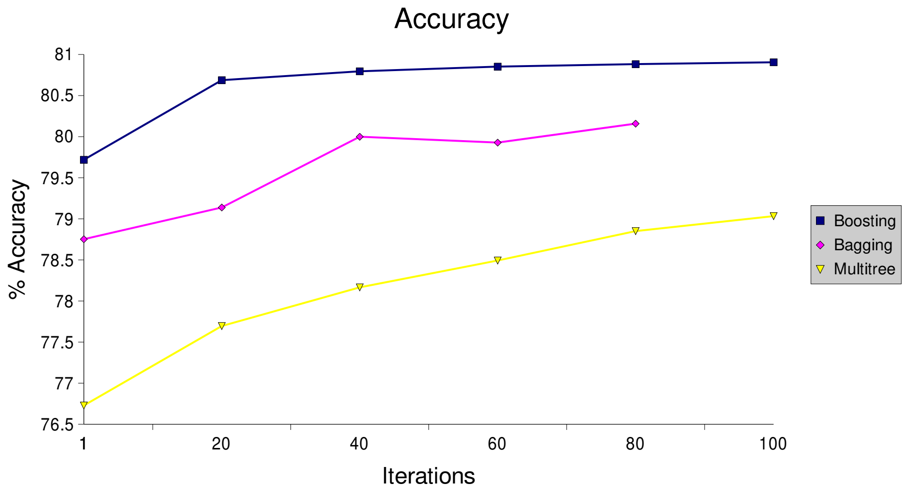


Figure 10.1: Accuracy obtained by the ensemble methods depending on the number of iterations.

The best results are obtained by *Boosting*. However, it does not increase significantly the accuracy with respect to the single classifier. *Bagging* gets slight lower results although the effect of the ensemble method seems to be more important. Note that the accuracy of *Bagging* with 100 classifiers is not in the figure; the cause of this absence is that the computer ran out of memory for such configuration. Finally, our method (Multi-tree) obtains lower results. However, the increase of accuracy of the ensemble method with raising the number of iterations is much greater than for the other two methods.

The comparison of the accuracy obtained by the ensemble methods is not very fair since the base classifier does not get similar accuracy (see the column of the single classifier in Table 10.14). Therefore, we consider that a better comparison can be performed if we consider only the datasets where *Bagging* and Multi-tree obtain similar accuracy for just one iteration. These datasets are all, except from 4, 8, 9, 17, 22. The latter are the datasets where the difference was greater than 4%. With this subset of the datasets, we have generated a figure for showing the accuracy of the multi-classifiers methods. This information is shown in Figure 10.2. Initially, the best results are obtained by *Boosting*, whereas *Bagging* and *Multi-tree* stay slightly lower. This is most probably because they do not use pruning. When the number of iterations of the ensemble methods is increased, all methods improve in accuracy as expected. In this case, we can see that *multi-tree* is the method that further enhances the results, it even overpasses *Boosting* at 80 iterations or more.

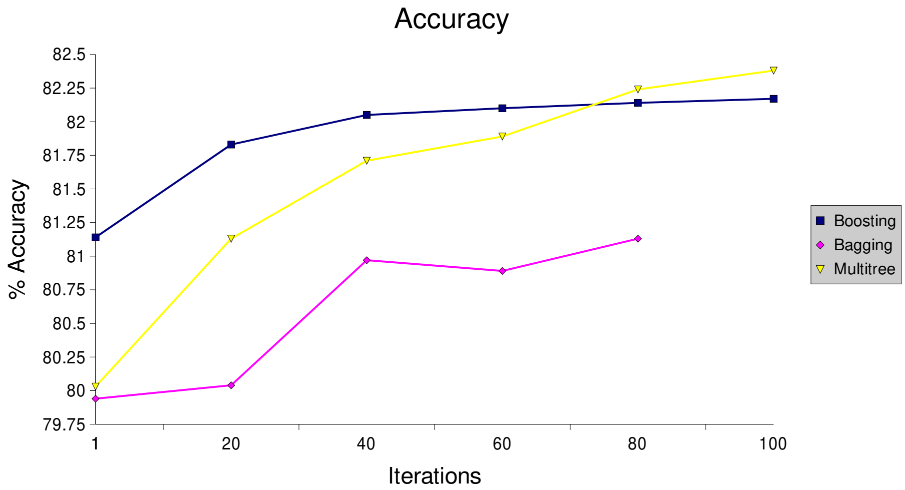


Figure 10.2: Accuracy obtained by the ensemble methods depending on the number of iterations (datasets 4,8,9,17,22 excluded).

Finally, although the shared ensemble method has suitable properties in terms of accuracy of the combined classifier, the most attractive feature of this algorithm is the sharing of some parts of the members of the ensemble. This fact permits a good dealing of computational resources. Figure 10.3 shows the average training time of all the datasets for *Bagging*, *Boosting*, and Multi-tree (shared ensemble) depending on the size of the ensemble. While *Bagging* and *Boosting*[2] present a linear increase of time, for the shared ensemble technique the increase is sub-linear. Note that the implementation features of the methods (Weka is implemented in Java, while SMILES is implemented in C++) clearly affect the training time. However, this does not produce significant changes in the asymptotic behaviour of the algorithms when varying the number of iterations.

10.4 Archetype Model of a Multi-tree

Comprehensibility has been the major advantage that has been advocated for supporting some machine learning methods such as decision tree learning, rule learners or ILP. A model that is not comprehensible may be useful to obtain good predictions, but it cannot provide knowledge about how predictions are made.

As we have stated in Chapter 6, accuracy is significantly increased with ensemble

[2]The observed non-linear behaviour of *Boosting* is due to a technique implemented in *Boosting* that stops the algorithm when accuracy does not improves further from iteration to iteration.

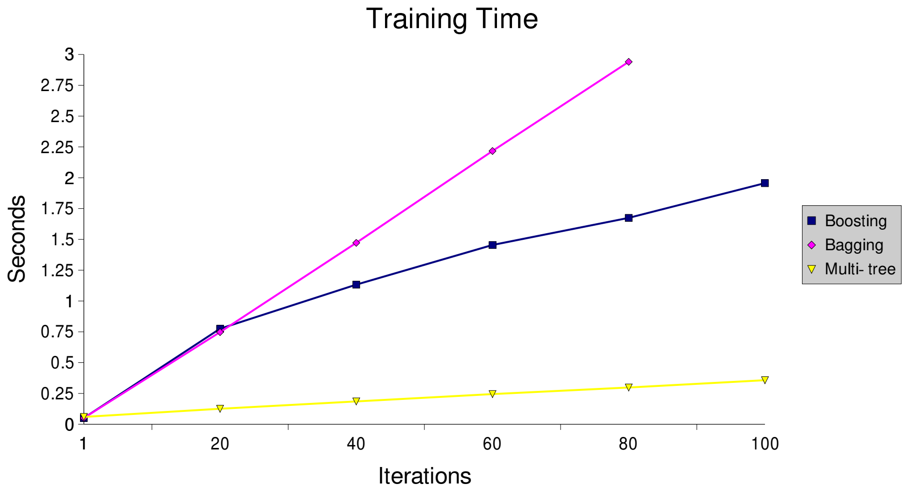


Figure 10.3: Training time required by the ensemble methods depending on the number of iterations.

methods. Nevertheless, they have some drawbacks, mainly the loss of comprehensibility of the combined model and the large amount of memory required to store the hypotheses.

One possible solution for solving most of these problems could be to reduce to one single hypothesis from the combination of m hypotheses without losing too much accuracy with respect to the combined hypothesis.

Instead of using classical methods for selecting one hypothesis, such as the hypothesis with the lowest expected error, or the one with the smallest size (Occam's razor), in this section we present a method to select the single hypothesis that is most similar to the combined hypothesis. This single hypothesis will be called an *archetype* or *representative* of the ensemble and can also be seen as an 'explanation' of the ensemble.

To do this, the main idea is to consider the combination as an oracle that would allow us to measure the similarity of each single hypothesis with respect to this oracle. More precisely, for a hypothesis or solution h and an unlabelled example e, let us define $h(e)$ as the class or label assigned to e by h. Consider an ensemble of solutions $\mathcal{E} = h_1, h_2, \cdots h_m$ and a method of combination χ. By $\Sigma_{\chi,\mathcal{E}}$ we denote the combined solution formed by using the method χ on $\mathcal{E}$. Thus, $\Sigma_{\chi,\mathcal{E}}(e)$ is the class assigned to e by the combined solution. Now, we can use $\Sigma_{\chi,\mathcal{E}}$ as an oracle, which, generally, gives better results than any single hypothesis [Dietterich, 2000a]. The question is to select a single hypothesis h_i from $\mathcal{E}$ such that h_i is the most similar (semantically) to the

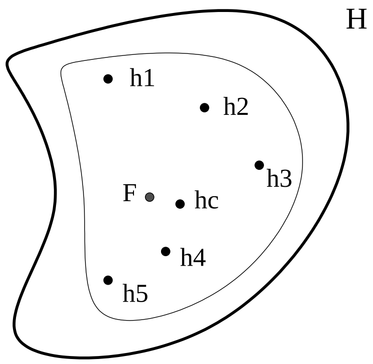


Figure 10.4: Representation of an ensemble of hypotheses

oracle $\Sigma_{\chi,\mathcal{E}}$.

This rationale is easy to understand following the representation used in a statistical justification for the construction of good ensembles presented by Dietterich in [Dietterich, 2000a]. A learning algorithm is employed to find different hypotheses $\{h_1, h_2, \cdots, h_m\}$ in the hypothesis space or language $\mathcal{H}$. By constructing an ensemble out of all these classifiers, the algorithm can "average" their votes and reduce the risk of choosing a wrong classifier. Figure 10.4 depicts this situation. The outer curve denotes the hypothesis space $\mathcal{H}$. The inner curve denotes the set of hypotheses that give a reasonably good accuracy on the training data and hence could be generated by the algorithm. The point labelled by F is the true hypothesis. If an ensemble h_c is constructed by combining the accurate hypotheses, then h_c will be a good approximation to F. However, h_c is an ensemble, which means that it needs to store $\{h_1, h_2, \cdots, h_5\}$ and it is not comprehensible. For this reason, we are interested in selecting the single hypothesis from $\{h_1, h_2, \cdots, h_m\}$ that would be closest to the combination h_c. In the situation described in Figure 10.4, we would select h_4 as the *archetype* or *representative* of the ensemble.

A final question, also pointed out by [Dietterich, 2000a], is that a statistical problem arises when the amount of training data available is too small compared to the size of the hypothesis space $\mathcal{H}$. The selection of a good archetype would not be possible if a sufficient amount of data is not available for comparing the hypotheses. The reserve of part of the training data is generally not a good option because it would yield a smaller training dataset and the ensemble would have a lower quality. This problem has a peculiar but simple solution: the generation of random unlabelled datasets.

Although the technique presented in this section is applicable to many kinds of ensemble methods, we will illustrate it with *shared* ensembles, because the number of hypotheses in this kind of structure grows exponentially with respect to the number of iterations of the learning algorithm. Therefore, there is a much bigger population which the representative can be extracted from.

10.4.1 Hypothesis Similarity

As we have stated, our proposal is to select the single hypothesis which is most similar to the combined one. Consequently, we must employ different measures of hypothesis similarity. These metrics and an additional dataset will lead to the estimation of a value of similarity between two hypotheses.

Some measures of hypothesis similarity (or diversity) have been introduced in Section 6.1.1 with the aim of obtaining an ensemble with high diversity. Although our goal is to find similar hypotheses rather than diverse ones, we can use the same metrics, at least those that are able to compare a pair of hypotheses. Precisely, some of the measures employed in [Kuncheva and Whitaker, 2002] are defined for a set of hypotheses and others for a pair of hypotheses. We are interested in the "pairwise diversity measures", since we want to compare a single hypothesis with an oracle. The approach presented by [Kuncheva and Whitaker, 2002] requires the correct class to be known. The additional dataset should be labelled, which means that part of the training set should be reserved for the estimation of similarities. Secondly, some other metrics are only applicable to two classes. As a result, in what follows, we can only use the pairwise metrics that can be estimated by using an unlabelled dataset and that can be used for problems with more than two classes.

Consequently, for our purpose we will employ the three measures defined in Section 6.1.1, i.e. θ, κ, and Q measures. The reason is that we can use these metrics to compare the similarity between two or more classifiers, with respect to an unlabelled reference dataset.

Obviously, the greater the reference dataset is, all of the previous metrics will give a better estimate of similarity. In our case, and since the previous measures use the contingency matrix, we can have huge reference datasets available: random invented datasets.

10.4.2 Random Invented Datasets

In many situations, a single hypothesis may be the one which is the most similar to the combined hypothesis *with respect to the training set*. However, it may not be the most similar one in general (*with respect to other datasets*). In some cases, e.g. if we do not use pruning, then all the hypotheses (and hence the combined solution) may have 100% accuracy with respect to the training set, and all the hypotheses are equally "good". Therefore, it is suitable or even necessary to evaluate similarity with respect to an external (and desirably large) reference dataset. However, we cannot always reserve part of the training set for this, or it could be counterproductive.

Then the idea is to use the entire training set to construct the hypotheses and to use a *random* dataset to select one of them. We consider that the examples in the training set are equations of the form $f(\cdots) = c$, where f is a function symbol and c is the class of the term $f(\cdots)$. Given a function f with a arguments, an unlabelled random example is any instance of the term $f(X_1, X_2, \cdots, X_a)$, i.e., any term of the form $f(v_1, v_2, \cdots, v_a)$ obtained by replacing every attribute X_i by values v_i from the attribute domain (attribute type). Note that an unlabelled random example is not an equation (a full example) because we include no information about the correct class.

Apparently, there are many ways to generate a random dataset. One possibility is to ignore any previous training set and just consider the types of each attribute (if known). Then we can use a uniform distribution for generating each of the attributes. One problem here arises with continuous attributes, because the *min* and *max* limits must be known. A second possibility is to use the training set as a reference and use the values of the attributes that appear in the training set.

We will use the following technique to generate each random unlabelled example: each attribute X_i of a new example is obtained as the value v_i in a different example $f(v_1, \ldots, v_i, \ldots, v_a)$ selected from the training set by using a uniform distribution. This procedure of generating instances assumes that all the attributes are independent, and just maintains the probabilities of appearance of the different values observed in each attribute of the training dataset. There is no need, then, of using kernel density estimation methods.

10.4.3 Selecting an Archetype from a Shared Ensemble

In a shared ensemble, we are not interested (because it would be unfeasible) to compute the similarity of each hypothesis with respect to the combined hypothesis, because there would be an exponential number of comparisons. What we are interested in is a measure of similarity for each node with respect to the combined solution, taking into account only the examples of the invented dataset that fall into a node.

The general idea is, once the multi-tree is constructed, to use its combination to predict the classes for the previously unlabelled invented dataset. Given an example e from the unlabelled invented dataset, this example will fall into different OR-nodes and finally into different leaves, giving different class vectors. Then, the invented dataset is labelled by voting these predictions in the way explained in Section 10.2.

The following step consists in the extraction of a single hypothesis from the multi-tree according to the similarity with respect to the combination hypothesis, and employing an invented dataset as reference.

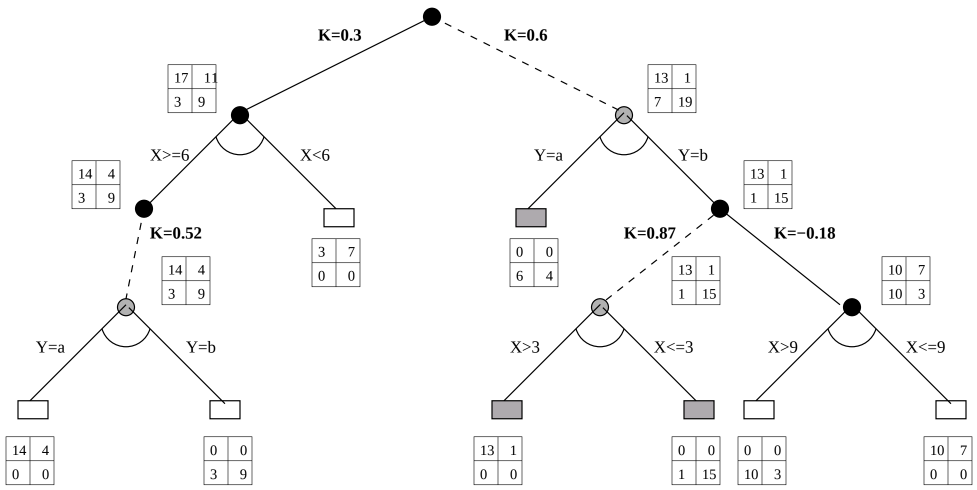


Figure 10.5: Selection of a single decision tree from the multi-tree structure.

Definition 10 (Archetype selection from a multi-tree) *Given an invented dataset labelled with the combined hypothesis, a contingency matrix M $(C \times C)$ is defined for each node (internal or leaf) of the multi-tree, where C is the number of classes. M is always initialised to 0. For each example in the labelled invented dataset, we increment the cell $M_{a,b}$ of each leaf where the example falls by 1, with a being the predicted class by the leaf and b being the predicted class by the combination. When all the examples have been evaluated and the matrices in the leaf nodes have been assigned, then we propagate the matrices upwards as follows:*

- *For the contingency matrix M of OR-nodes, we accumulate the contingency matrix of their m children nodes: $(M_1 + M_2 + \cdots + M_m)$.*
- *For the contingency matrix M of AND-nodes, the node of their children with greater κ (or other similarity measure) is selected and its matrix is propagated upwards. The selected node is marked.*

This ultimately generates a hypothesis that is hopefully the most similar to the combined hypothesis, using a particular invented dataset and a given similarity measure.

Figure 10.5 shows the selection of one hypothesis from a multi-tree according to the contingency matrix. The AND-nodes are represented with an arc. The leaves are represented by rectangles. First, we fill the matrices of the leaves. Then, we propagate these upwards as has been detailed previously. Finally, when we reach the top of the tree, it is straightforward to extract the solution by simply descending the multi-tree by the marked nodes. In the figure, the marked nodes are represented by the dashed lines, and the leaves of the selected hypothesis are shadowed.

Let us study the properties of our selection technique, depending on the similarity measure. Let us first consider the θ measure. In this case, we can demonstrate that we select the single tree of the multi-tree that is most similar to the combination. To prove this, we only have to prove that if we select the option with greater θ in an OR-node, then adding the other "siblings" to the selection the option will still be optimal.

Theorem 11 *The selection procedure presented in Definition 10 extracts the most similar solution to its combination according to a random dataset and using θ as similarity measure.*

Proof. Let us denote by d the depth of the tree, namely the maximum number of levels of OR-nodes in the multi-tree. We can prove the theorem by induction on the depth of the multi-tree.

Base Case: d=1

If d=1, the multi-tree will be formed by an AND-node at the top, its OR-nodes children, and for each OR-node there will be a set of leaf AND-nodes.

For every leaf node, a contingency matrix M is computed according to the examples that fall into that leaf. For each OR-node, the contingency matrix M is computed by adding the contingency matrices of their children, since the children AND-nodes are complementary, and therefore the examples will be different in each leaf. We compute the similarity of the OR-node with respect to the combination using the θ measure.

Finally, the contingency matrix of the child OR-node with the greatest θ is assigned to the top AND-node, i.e. the OR-node most similar to the combination of the multi-tree according to the θ measure. Finally, we mark this node to be extracted later. It is clear that this node is the most similar one.

Induction Case: d=$n+1$

Let $A^{\{n+1\}}$ be the top AND-node of the level $n+1$ with m OR-node children: $O^{\{n+1\}}_{\{1\}}$, $O^{\{n+1\}}_{\{2\}}$,...,$O^{\{n+1\}}_{\{m\}}$. The OR-node $O^{\{n+1\}}_{\{i\}}$ has h AND-node children. Let $A^n_{\{i,j\}}$,..., $A^n_{\{i,h\}}$ be the AND-nodes at level n below node $O^{\{n+1\}}_{\{i\}}$. Let $A^n_{\{i,j\}}$ be a node with l OR-node children: $O^n_{\{i,j,1\}}$, $O^n_{\{i,j,2\}}$,...,$O^n_{\{i,j,l\}}$. This structure can be seen in Figure 10.6.

If $O^n_{\{i,j,k\}}$ is selected as the most similar to the combination for $A^{\{n\}}_{\{i,j\}}$, (level n) we have to prove that in level $n+1$ the selection of level n is still the best selection, i.e. there is no other local choice that would be better for the global choice.

Let $M^{\{n\}}_{\{i,j,k\}}$ be the contingency matrix of node $O^{\{n\}}_{\{i,j,k\}}$, and $\theta^{\{n\}}_{\{i,j,k\}}$ the value

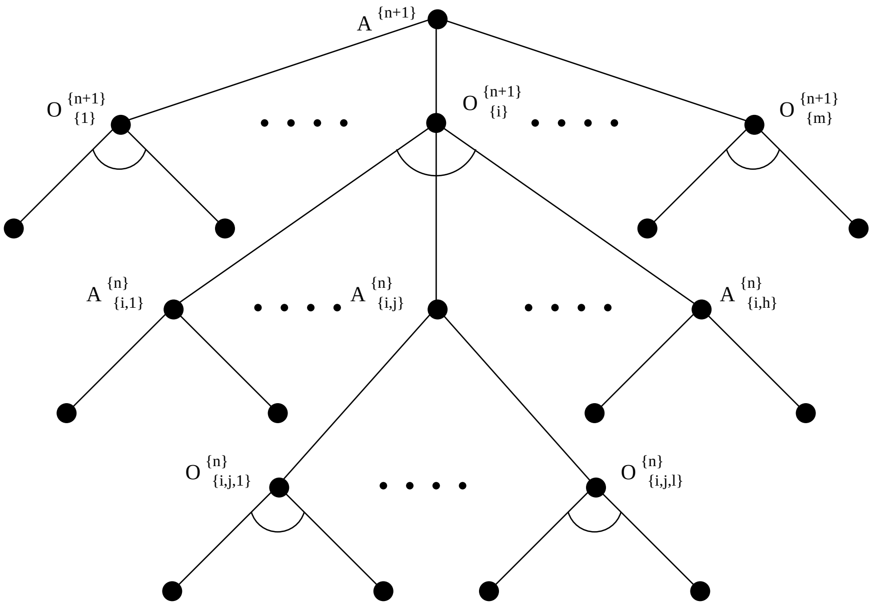


Figure 10.6: Structure of the decision multi-tree.

computed for that node. Since it has been selected, it holds that :

$$\theta^{\{n\}}_{\{i,j,k\}} \geq \theta^{\{n\}}_{\{i,j,p\}}, \forall 1 \leq p \leq l$$

Therefore, we have to prove that:

$$\theta^{\{n+1\}}_{\{i^{(j,k)}\}} \geq \theta^{\{n+1\}}_{\{i^{(j,p)}\}}, \forall 1 \leq p \leq l, p \neq k$$

where $\theta^{\{n+1\}}_{\{i^{(j,k)}\}}$ represents the fact that node $A^{\{n\}}_{\{i,j\}}$ has selected $O^{\{n\}}_{\{i,j,k\}}$.

According to the definition of θ for a matrix with C classes (see Section 6.1.1):

$$\theta = \sum_{i=1}^{C} \frac{M_{i,i}}{e} = \frac{x}{x+y}$$

where $x = \sum_{i=1..C} M_{i,i}$, $y = \sum_{i=1..C,j=1..C,i\neq j} M_{i,j}$,
and $x + y = \sum_{i=1..C,j=1..C,i} M_{i,j} = e$. Then,

$$\theta^{\{n\}}_{\{i,j,k\}} = \frac{x^{\{n\}}_{\{i,j,k\}}}{x^{\{n\}}_{\{i,j,k\}} + y^{\{n\}}_{\{i,j,k\}}}$$

If we have that the inequality holds at level n:

$$\frac{x^{\{n\}}_{\{i,j,k\}}}{x^{\{n\}}_{\{i,j,k\}} + y^{\{n\}}_{\{i,j,k\}}} \geq \frac{x^{\{n\}}_{\{i,j,p\}}}{x^{\{n\}}_{\{i,j,p\}} + y^{\{n\}}_{\{i,j,p\}}}, \forall 1 \leq p \leq l, p \neq k \qquad (1)$$

since we are comparing OR-nodes, the same examples will fall in all of them, i.e.:

$$x^{\{n\}}_{\{i,j,a\}} + y^{\{n\}}_{\{i,j,a\}} = x^{\{n\}}_{\{i,j,b\}} + y^{\{n\}}_{\{i,j,b\}}, \forall 1 \leq a \leq l, 1 \leq b \leq l$$

Then, formula (1) is simplified as:

$$x^{\{n\}}_{\{i,j,k\}} \geq x^{\{n\}}_{\{i,j,p\}}, \forall 1 \leq p \leq l, p \neq k \qquad (2)$$

In node $O^{\{n+1\}}_{\{i\}}$, θ will be the addition of all its AND-node children:

$$\theta^{\{n+1\}}_{\{i^{(j,k)}\}} = \frac{x^{\{n\}}_{\{i,j,k\}} + x^{\{n\}}_{\{i^{-j}\}}}{x^{\{n\}}_{\{i,j,k\}} + x^{\{n\}}_{\{i^{-j}\}} + y^{\{n\}}_{\{i,j,k\}} + y^{\{n\}}_{\{i^{-j}\}}}$$

where

$$x^{\{n\}}_{\{i^{-j}\}} = \sum_{1 \leq t \leq h, t \neq j} x^{\{n\}}_{\{i,t\}}$$

represents all except j.

Finally, we want to prove that

$$\theta^{\{n+1\}}_{\{i^{(j,k)}\}} \geq \theta^{\{n+1\}}_{\{i^{(j,p)}\}} \forall 1 \leq p \leq l$$

$$\frac{x^{\{n\}}_{\{i,j,k\}} + x^{\{n\}}_{\{i^{-j}\}}}{x^{\{n\}}_{\{i,j,k\}} + x^{\{n\}}_{\{i^{-j}\}} + y^{\{n\}}_{\{i,j,k\}} + y^{\{n\}}_{\{i^{-j}\}}} \geq \frac{x^{\{n\}}_{\{i,j,p\}} + x^{\{n\}}_{\{i^{-j}\}}}{x^{\{n\}}_{\{i,j,p\}} + x^{\{n\}}_{\{i^{-j}\}} + y^{\{n\}}_{\{i,j,p\}} + y^{\{n\}}_{\{i^{-j}\}}}$$

$$, \forall 1 \leq p \leq l, p \neq k \quad (3)$$

since we are comparing OR-nodes, the same examples fall in all of them, i.e.

$$x^{\{n\}}_{\{i,j,k\}} + x^{\{n\}}_{\{i^{-j}\}} + y^{\{n\}}_{\{i,j,k\}} + y^{\{n\}}_{\{i^{-j}\}} = x^{\{n\}}_{\{i,j,p\}} + x^{\{n\}}_{\{i^{-j}\}} + y^{\{n\}}_{\{i,j,p\}} + y^{\{n\}}_{\{i^{-j}\}}$$

$$, \forall 1 \leq p \leq l$$

then, formula (3) is simplified as :

$$x^{\{n\}}_{\{i,j,k\}} + x^{\{n\}}_{\{i^{-j}\}} \geq x^{\{n\}}_{\{i,j,p\}} + x^{\{n\}}_{\{i^{-j}\}}, \forall 1 \leq p \leq l$$

which holds because of the induction hypothesis (formula (2)). This means that Definition 10 is indeed a correct procedure for obtaining the hypothesis that is the most similar to the combination, taking θ as a similarity metric. □

This interesting property does not hold when using κ as similarity measure, as the following counter example demonstrates:

Example 16 *Consider a multi-tree for a two-class problem with an OR-node O with two AND-nodes children: A_1 and A_2. The AND-node A_2 is a leaf with a contingency matrix:*

$$M_2 = \begin{bmatrix} 10 & 1 \\ 5 & 5 \end{bmatrix}$$

On the other hand, A_1 has two children OR-nodes: $O_{\{1,1\}}, O_{\{1,2\}}$. These nodes have the following contingency matrices:

$$M_{\{1,1\}} = \begin{bmatrix} 3 & 1 \\ 2 & 2 \end{bmatrix} \quad M_{\{1,2\}} = \begin{bmatrix} 1 & 1 \\ 2 & 4 \end{bmatrix}$$

Since these are two sibling OR-nodes, they have the same number of examples: 8. Our method selects the OR-node that is the most similar one according to the κ measure computed from the contingency matrices. In this case: $\kappa_{\{1,1\}} = 0.25$, and $\kappa_{\{1,2\}} = 0.1428$. Therefore, the method will select $O_{\{1,1\}}$ as the most similar to the combination.

According to this selection, the contingency matrix of O will be:

$$M_O = \begin{bmatrix} 13 & 2 \\ 7 & 7 \end{bmatrix}$$

And the corresponding κ will be $\kappa = 0.37108$.

However, if we select the other OR-node ($O_{\{1,2\}}$), the contingency matrix of O would be:

$$M_O = \begin{bmatrix} 11 & 2 \\ 7 & 9 \end{bmatrix}$$

And its $\kappa = 0.3944$, which is greater than the one given by the original selection. Then, the procedure (Definition 10) does not always select the single decision tree with the greatest κ. This example also shows that a correct procedure would have an exponential cost. For this reason, although the procedure of Definition 10 is approximate for κ, we will still use it for the sake of efficiency.

Therefore, we can summarise the approach in five different steps:

1. **Multi-tree generation**: The first step consists in the generation of a multi-tree from a training dataset.

2. **Multi-tree combination**: The invented dataset is labelled by the combination of the shared ensemble. A method of combination of hypotheses can be specified.

3. **Invented dataset**: In this phase, an unlabelled invented dataset is randomly created.

4. **Calculation and propagation of contingency matrices**: A contingency matrix is assigned to each node of the multi-tree, using the labelled invented dataset and a similarity metric.

5. **Selection of a solution**: An archetype hypothesis is extracted from the multi-tree by descending the multi-tree through the marked nodes.

10.4.4 Experiments

In this section, we present an experimental evaluation of our approach. We have included the archetype method in the SMILES system [Ferri *et al.*, 2002d]. For the experiments, we used GainRatio [Quinlan, 1993] as splitting criterion. We chose a random method (see Section 10.3) for populating the shared ensemble (after a solution is found, a suspended OR-node is woken at random) and we used the *maximum* strategy for combination.

Since there are many sources of randomness, we have performed the experiments by averaging 10 results of a 10-fold cross-validation. This makes a total of 100 runs (each one with a different multi-tree construction, random dataset and hypothesis selection process) for each pair of method and dataset.

In the following, we will use the following notation:

- *First Solution*: the solution given by just one hypothesis (the first hypothesis that is obtained). Therefore, this is similar to the solution given by C4.5 [Quinlan, 1993].

- *Combined Solution*: the solution given by combining the hypotheses of the ensemble (in our case, the multi-tree, as described in Section 10.2).

- *Archetype Solution*: the single solution which is most similar to the combined solution computed by the archetype method.

- *Occam Solution*: the single solution with the lowest number of rules, i.e., the shortest solution.

Our goal is to show that a significant gain of accuracy and AUC can be obtained from the *First Solution* to the *Archetype* and *Occam* methods as long as the size of the ensemble increases. Another question to be answered is to determine which method to extract a single solution from an ensemble is better: *Archetype* or *Occam*. But first we must explore the several options for the extraction of the archetype.

Evaluating Similarity Metrics

Table 10.16 shows the accuracy for each dataset and the geometric means of all of them, depending on the similarity measure employed. Table 10.17 exhibits the results in terms of AUC. We study three different similarity metrics: κ, θ and Q. The multi-tree has been generated exploring 100 suspended OR-nodes.

The results show that the *Archetype* method is very dependent on the measure of similarity used: in terms of accuracy, θ seems to be the best metric and Q the worst. We do not know whether this result is given because θ is indeed the best metric or because using θ Theorem 11 holds. Regarding AUC, the results obtained using the κ measure are the best, followed by θ, and finally Q. We do not know why different results are obtained with AUC. Maybe a more thorough experimental evaluation (using t-tests) could confirm or refuse this behaviour. In the following, we will use the θ measure.

Influence of the Size of the Invented Dataset

Similarity is approximated through the use of an invented dataset. Let us study the influence of its size, varying from 10 to 100,000 examples. The similarity metric and the size of the multi-tree are fixed to θ and 100 alternative opened OR-trees, respectively.

Table 10.18 shows the accuracy and Table 10.19 the AUC value. The conclusion that can be inferred from these tables is that in order to obtain a good archetype hypothesis, the similarity metric has to be computed as accurately as possible and for this purpose, the size of the invented dataset is crucial.

Influence of the Size of the Ensemble

The effect of the size of the multi-tree is evaluated in Table 10.20 (accuracy) and Table 10.21 (AUC). In these tables, we show the accuracy of the first single solution and the accuracy of the combination, the archetype solution and the Occam solution for multi-trees created by exploring 50, 100, and 200 alternative OR-nodes.

The results in accuracy are quite encouraging: by simply exploring 50 OR-nodes, the archetype solution surpasses the first solution and the Occam solution. This difference is increased as long as the multi-tree is populated. This is mainly due to the improvement in the accuracy of the combined solution and the fact that the archetype hypothesis can actually get close to it. The Occam solution does not seem to improve the accuracy as much. Nevertheless, the Occam hypothesis can also be regarded as a way to obtain more and more compact solutions without losing accuracy (see section 9.3.2).

Regarding AUC, the results are not very satisfactory, the archetype solution never improves the AUC of the first solution. Moreover, the Occam solution obtains more AUC than the archetype solution in general. We opine that the cause of this poor behaviour is the configuration of the archetype method with respect to accuracy. For instance, we are employing θ as similarity measure, while, as we have seen in

#	Arc. κ	Arc. Q	Arc. θ
1	79,82	79,73	79,82
2	93,91	94,01	94,14
3	92,23	92,45	92,00
4	89,78	89,45	89,78
5	99,47	99,47	99,48
6	48,35	47,96	48,27
7	90,92	90,81	90,64
8	38,31	39,54	38,31
9	66,27	66,20	66,67
10	76,87	76,50	76,75
11	51,57	51,28	51,62
12	76,47	76,13	77,53
13	63,33	62,36	63,69
14	78,67	78,28	79,08
15	94,14	94,72	94,21
16	93,47	94,00	93,60
17	64,56	64,65	64,79
18	99,91	99,91	99,91
19	73,58	72,55	73,58
20	97,78	97,78	97,78
21	93,43	92,71	93,67
22	70,09	69,95	70,17
23	60,00	59,00	60,12
24	95,87	95,84	95,89
25	61,67	61,47	61,20
26	99,20	99,24	99,23
27	77,09	77,20	77,39
28	92,24	92,29	92,12
Geomean	77,13	77,01	77,22

Table 10.16: Accuracy comparison between measures of similarity.

#	Arc. κ	Arc. Q	Arc. θ
1	0,816	0,817	0,817
2	0,976	0,981	0,980
3	0,967	0,967	0,964
4	0,949	0,935	0,934
5	1,000	0,999	0,999
6	0,657	0,655	0,656
7	0,994	0,987	0,989
8	0,630	0,588	0,622
9	0,645	0,650	0,647
10	0,887	0,902	0,900
11	0,666	0,658	0,668
12	0,763	0,758	0,760
13	0,729	0,706	0,710
14	0,824	0,823	0,825
15	0,982	0,984	0,983
16	0,989	0,985	0,985
17	0,657	0,670	0,669
18	1,000	1,000	1,000
19	0,677	0,742	0,755
20	0,991	0,988	0,988
21	0,973	0,967	0,969
22	0,761	0,764	0,765
23	0,573	0,419	0,417
24	0,997	0,997	0,996
25	0,760	0,765	0,765
26	0,998	0,997	0,997
27	0,882	0,870	0,870
28	0,976	0,977	0,972
Geomean	0,834	0,824	0,827

Table 10.17: AUC comparison between measures of similarity.

		10	100	1000	10000	100000
#	Comb	Arc	Arc	Arc	Arc	Arc
1	82,06	77,85	77,53	79,40	79,94	79,84
2	95,01	94,30	93,97	93,65	94,19	94,04
3	93,45	92,39	91,75	91,98	92,00	92,86
4	91,52	78,65	82,52	87,60	89,78	89,99
5	99,34	99,45	99,47	99,46	99,47	99,43
6	49,52	48,07	47,52	48,10	48,43	48,50
7	94,22	89,67	90,47	91,19	90,72	91,22
8	37,08	37,15	38,23	37,23	37,77	37,23
9	68,23	65,23	66,20	66,87	66,93	66,33
10	77,25	73,94	75,56	76,56	76,75	76,19
11	54,76	51,76	51,47	51,78	51,63	52,86
12	81,80	74,93	77,93	78,20	78,00	78,67
13	67,47	63,06	63,86	64,08	64,75	64,58
14	83,33	77,56	78,08	77,64	79,08	79,56
15	95,56	93,84	94,72	94,77	94,21	94,67
16	94,80	92,60	93,80	93,00	93,80	93,67
17	67,76	63,62	63,53	65,32	64,76	65,12
18	99,78	98,73	99,55	99,84	99,91	99,80
19	76,65	70,28	71,27	73,23	73,73	73,93
20	97,71	97,22	97,33	97,71	97,78	97,76
21	92,95	92,52	92,19	93,24	93,33	93,86
22	72,74	69,54	69,70	68,87	69,33	69,71
23	62,13	56,50	58,50	61,38	60,88	62,50
24	96,31	95,63	95,72	95,71	95,91	95,95
25	62,40	61,87	61,93	61,80	61,47	61,67
26	99,31	99,12	99,18	99,16	99,17	99,20
27	81,95	76,26	76,05	77,57	77,41	77,75
28	93,29	92,12	91,65	91,47	92,35	92,94
Geomean	78,95	75,84	76,44	77,06	77,29	77,52

Table 10.18: Influence in accuracy of the size of the invented dataset.

		10	100	1000	10000	100000
#	Comb	Arc	Arc	Arc	Arc	Arc
		10	100	1000	10000	100000
1	0.800	0.821	0.820	0.818	0.824	0.825
2	0.985	0.978	0.978	0.979	0.979	0.979
3	0.980	0.966	0.966	0.964	0.964	0.968
4	0.937	0.827	0.874	0.921	0.934	0.939
5	0.999	0.999	0.999	0.999	0.999	0.999
6	0.678	0.654	0.652	0.657	0.660	0.659
7	0.997	0.986	0.989	0.989	0.990	0.989
8	0.640	0.574	0.625	0.607	0.618	0.611
9	0.663	0.634	0.632	0.637	0.641	0.634
10	0.941	0.914	0.900	0.899	0.900	0.897
11	0.690	0.667	0.663	0.669	0.665	0.676
12	0.819	0.750	0.778	0.769	0.761	0.742
13	0.787	0.718	0.729	0.724	0.727	0.716
14	0.857	0.813	0.818	0.811	0.825	0.831
15	0.989	0.982	0.985	0.984	0.983	0.983
16	0.994	0.982	0.986	0.984	0.984	0.985
17	0.706	0.666	0.666	0.670	0.670	0.668
18	1.000	0.994	0.998	0.998	0.999	0.999
19	0.758	0.702	0.719	0.747	0.759	0.759
20	0.991	0.988	0.990	0.988	0.988	0.988
21	0.990	0.972	0.969	0.971	0.972	0.980
22	0.791	0.761	0.762	0.753	0.755	0.762
23	0.371	0.441	0.455	0.444	0.413	0.456
24	0.998	0.996	0.996	0.996	0.996	0.997
25	0.786	0.770	0.772	0.771	0.773	0.767
26	0.999	0.996	0.997	0.997	0.997	0.997
27	0.913	0.861	0.860	0.872	0.875	0.875
28	0.994	0.976	0.973	0.972	0.973	0.979
GeoMean	0.842	0.819	0.826	0.827	0.827	0.830

Table 10.19: Influence in AUC of the size of the invented dataset.

	1	50			100			200		
#	1st	Comb	Arc	Occ	Comb	Arc	Occ	Comb	Arc	Occ
1	77.94	80.39	78.95	77.61	82.06	79.84	77.53	83.79	80.94	77.35
2	93.81	94.84	93.83	93.75	95.01	94.04	94.04	95.16	93.68	93.35
3	92.43	93.07	92.48	92.59	93.45	92.86	92.39	92.91	92.68	92.50
4	88.61	90.98	89.61	88.76	91.52	89.99	88.77	91.54	89.64	88.74
5	99.61	99.45	99.49	99.60	99.34	99.43	99.61	99.30	99.47	99.61
6	48.29	48.90	48.14	48.37	49.52	48.50	47.97	49.99	48.05	47.82
7	91.47	94.31	90.94	92.36	94.22	91.22	93.14	94.31	91.17	93.22
8	38.23	36.92	37.69	38.69	37.08	37.23	37.85	37.08	37.31	40.46
9	66.33	67.53	66.10	64.97	68.23	66.33	65.87	67.70	66.47	65.70
10	74.06	77.25	76.19	73.81	77.25	76.19	73.81	77.25	76.19	73.81
11	52.08	53.91	51.64	52.13	54.76	52.86	52.12	54.49	52.38	51.79
12	75.80	81.27	77.93	74.87	81.80	78.67	75.20	81.73	78.93	75.93
13	62.72	65.39	62.89	62.56	67.47	64.58	62.78	68.28	65.17	64.42
14	78.53	82.33	78.97	78.97	83.33	79.56	79.14	83.22	80.00	78.56
15	94.74	95.67	94.56	94.09	95.56	94.67	94.37	95.67	94.05	93.98
16	94.13	94.93	93.80	94.20	94.80	93.67	94.27	95.53	94.40	94.60
17	64.44	66.82	63.76	65.00	67.76	65.12	64.12	68.24	64.59	64.47
18	95.22	99.44	99.55	99.47	99.78	99.80	99.96	99.95	99.95	100.00
19	70.97	73.52	71.43	69.87	76.65	73.93	69.82	78.43	75.80	69.53
20	97.47	97.84	97.89	97.55	97.71	97.76	97.60	97.64	97.62	97.62
21	91.95	92.95	93.76	93.00	92.95	93.86	93.05	91.81	93.38	93.43
22	69.63	72.39	69.49	69.42	72.74	69.71	69.01	73.91	70.28	69.45
23	62.25	62.13	61.88	63.13	62.13	62.50	64.75	62.13	62.63	64.75
24	95.91	96.30	95.95	96.02	96.31	95.95	96.11	96.31	95.87	95.98
25	60.60	61.73	61.27	60.93	62.40	61.67	60.60	64.13	62.07	61.20
26	99.23	99.32	99.25	99.27	99.31	99.20	99.24	99.26	99.18	99.27
27	77.16	80.22	77.36	77.43	81.95	77.75	78.16	83.40	78.16	78.43
28	93.00	92.59	90.82	92.71	93.29	92.94	93.41	93.18	91.94	92.53
GMean	76.73	78.33	76.99	76.87	78.95	77.52	76.94	79.26	77.61	77.19

Table 10.20: Influence in accuracy of the size of the multi-tree.

previous experiments, κ obtains better results in AUC. Thus, we think that, with a correct configuration of the archetype method, this can also be useful to obtain comprehensible classifiers with high AUC. Another option could be to define similarity metrics that could be designed for AUC rather than for accuracy.

10.4.5 Archetyping as the Last Layer for Hybrid Methods

The archetyping technique provides a means to extract a single, accurate and comprehensible solution from an ensemble of classifiers. We have used the combination of our multi-tree as our "oracle", i.e. the archetype should be similar to the combined solution.

An additional technical application arises if we consider any *external* oracle. In many situations, we have originally incomprehensible machine learning methods, such as neural networks [Lau, 1992] or support vector machines [Evgeniou and Pontil, 2001], or we have hybrid or ensemble methods that turn symbolic methods incomprehensible, such as boosting decision trees. If comprehensibility is a premise, all of these hybrid

	1	50			100			200		
#	1st	Comb	Arc	Occ	Comb	Arc	Occ	Comb	Arc	Occ
1	0.824	0.807	0.821	0.824	0.800	0.824	0.820	0.807	0.821	0.824
2	0.979	0.984	0.980	0.978	0.985	0.979	0.976	0.984	0.980	0.978
3	0.968	0.977	0.968	0.970	0.980	0.964	0.967	0.977	0.968	0.970
4	0.953	0.942	0.939	0.949	0.937	0.934	0.949	0.942	0.939	0.949
5	1.000	0.999	0.999	1.000	0.999	0.999	1.000	0.999	0.999	1.000
6	0.657	0.669	0.657	0.658	0.678	0.660	0.655	0.669	0.657	0.658
7	0.991	0.997	0.989	0.993	0.997	0.990	0.992	0.997	0.989	0.993
8	0.622	0.643	0.621	0.614	0.640	0.618	0.637	0.643	0.621	0.614
9	0.641	0.659	0.641	0.632	0.663	0.641	0.646	0.659	0.641	0.632
10	0.900	0.942	0.897	0.886	0.941	0.900	0.887	0.942	0.897	0.886
11	0.667	0.683	0.669	0.668	0.690	0.665	0.662	0.683	0.669	0.668
12	0.718	0.807	0.759	0.761	0.819	0.761	0.764	0.807	0.759	0.761
13	0.724	0.769	0.720	0.711	0.787	0.727	0.719	0.769	0.720	0.711
14	0.823	0.853	0.820	0.823	0.857	0.825	0.824	0.853	0.820	0.823
15	0.987	0.988	0.983	0.984	0.989	0.983	0.982	0.988	0.983	0.984
16	0.987	0.992	0.984	0.988	0.994	0.984	0.989	0.992	0.984	0.988
17	0.667	0.700	0.652	0.669	0.706	0.670	0.674	0.700	0.652	0.669
18	0.987	1.000	0.998	0.999	1.000	0.999	1.000	1.000	0.998	0.999
19	0.710	0.736	0.731	0.681	0.758	0.759	0.678	0.736	0.731	0.681
20	0.991	0.992	0.991	0.991	0.991	0.988	0.991	0.992	0.991	0.991
21	0.970	0.989	0.973	0.971	0.990	0.972	0.976	0.989	0.973	0.971
22	0.761	0.783	0.762	0.762	0.791	0.755	0.762	0.783	0.762	0.762
23	0.559	0.418	0.460	0.574	0.371	0.413	0.594	0.418	0.460	0.574
24	0.997	0.998	0.997	0.997	0.998	0.996	0.997	0.998	0.997	0.997
25	0.763	0.777	0.763	0.764	0.786	0.773	0.762	0.777	0.763	0.764
26	0.998	0.998	0.997	0.998	0.999	0.997	0.998	0.998	0.997	0.998
27	0.873	0.898	0.873	0.874	0.913	0.875	0.881	0.898	0.873	0.874
28	0.976	0.991	0.972	0.977	0.994	0.973	0.974	0.991	0.972	0.977
GMean	0.833	0.841	0.828	0.833	0.842	0.827	0.836	0.841	0.828	0.833

Table 10.21: Influence in AUC of the size of the multi-tree.

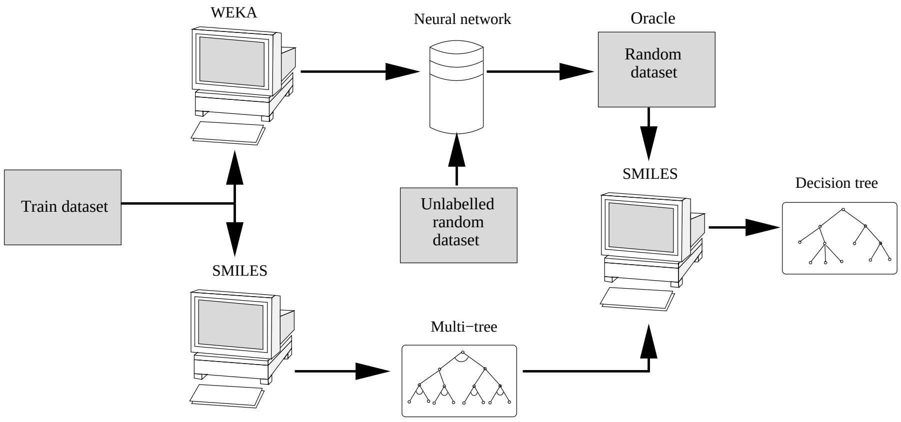


Figure 10.7: Representation of the selection of a decision tree by an external oracle

methods could not be used. However, all the potential of hybrid methods would be lost. We have seen that our multi-tree structure and the archetyping method provide a way to take the best of both approaches.

The idea is that SMILES, and its archetyping technology, can be placed as the final layer (or a "bypass" layer) to any hybrid or multi-layered classifier. SMILES would construct trees taking the original hybrid classifier as the oracle. Therefore, this could be regarded as a new method to "convert" incomprehensible models (neural networks, hybrid methods, cascade combinations, SVM, etc.) to comprehensible decision trees (with possibly a slight loss in accuracy/AUC). This approach is much more flexible than any other rule extraction method because it can be applied to any black-box source and any comprehensible destination. Figure 10.7 shows the described process where a neural network is learned (e.g. by using the WEKA system). In parallel, the SMILES system learns a multi-tree. The neural network is used to label an invented dataset and the final labelled dataset is input to SMILES to select the single hypothesis from the multi-tree that is most similar to the neural network classifier.

In the figure, SMILES also uses the original training set, but this would not be necessary in general. It is important to clarify that our proposal is quite different to stacking. Stacking and other hybrid or multi-layered combination techniques have as a result a combined model whose attributes are the outputs of other models, i.e. they are defined in terms of other model outputs. For instance, a decision tree as the final layer of a stacking or cascade combination framework does not yield a comprehensible model. On the other hand, archetyping generates a model that is defined in terms of the original attributes, and, hence, it can be comprehensible.

10.5 Conclusions and Contributions of this Chapter

This chapter has presented an ensemble method whose main feature is the use of a multi-tree that permits sharing common parts of the single components of the ensemble. The multi-tree structure is very similar to an option tree defined by Buntine in [Buntine, 1990, 1993]. Option trees compute the prediction of examples by combining by Bayes averaging the predictions of the option subtrees (OR-nodes). In [Kohavi and Kunz, 1997] the combination is made by majority vote. The main difference of options trees and our proposal is the process of construction of the tree. While options trees are constructed by a partially search, we employ a beam search to populate the multi-tree. Another relevant difference is that the option trees have only been tested with two fusion methods (maximum and average), while we have studied several fusion and transformation methods.

In fact, in this chapter, several combination methods or fusion strategies have been presented, as well as new class vector transformation techniques. The effectiveness of these methods has also been examined by an experimental evaluation. We have also investigated the importance of the size of the multi-tree with respect to the quality of the results obtained.

We have compared the new ensemble method with some well-known ensemble methods, namely *Boosting* and *Bagging*. The accuracy results for the new method are quite encouraging: although our results are initially worse than the other two methods, when the number of iterations is increased, the new approach equals and even excels the other methods. Nevertheless, as we have shown, it is in the use of resources where the shared ensembles offer an important advance. Due to the sharing of parts, much less time is required than with classical ensemble approaches to perform the same number of iterations. Therefore, our approach is very appropriate for complex problems where other ensemble methods such as *Boosting* or *Bagging* require huge amounts of memory and time.

We construct the multi-tree by a random extraction from the node list. Therefore, our technique is also related to randomisation [Dietterich, 2000b]. This technique constructs an ensemble of decision trees by injecting randomness in the splitting criterion. The idea is selecting randomly among the top 20 best splits. In this way, the method generates a different decision tree for each iteration. However, randomisation builds forests, i.e. an ensemble of completely independent decision trees.

On the other hand, we have presented a novel method for extracting a single solution from an ensemble of solutions without removing training data for validation. The most closely related work is Quinlan's miniboosting [Quinlan, 1998b]. However, Quinlan's method can be considered an ensemble method which generates three trees, followed by a merging stage where a single but quite complex tree could be obtained. Moreover, as he recognises, "although it is (usually) possible to construct a single merged decision tree that induces the same partition as a small ensemble, the tree is so large that it conveys no insight. This is a pity, as insight was the prime motivation for producing a single tree".

We overcome the previous problem by producing a single tree that is based on a hypothesis selection, using the combination as an oracle. Consequently, the result is a single comprehensible solution, an archetype or representative of the ensemble. As we have shown, the single solution obtained by our method is not 100% equivalent to the combination, but in general it gets reasonably close.

Moreover, it is clear that a similar technique could be used for regression models, using, e.g., minimum squared error as a discrepancy (similarity) metric.

From a more general point of view, ensemble methods have been used as an argument in favour of Epicurus' criterion (all consistent models should be retained) and against Occam's razor, because complex combined hypotheses usually obtain better results than the simplest solution [Webb, 1996]. A counterargument may be that the combination is usually expressed outside the hypothesis language. With our work, we have shown that even inside the hypothesis language, the shortest solution is not the best one.

Finally, it is important to clarify that an archetype solution cannot be obtained without an ensemble, and the quality of the representative would depend on the number of individual hypotheses in the ensemble. Note that this number is exponentially increased by the use of shared ensembles, in particular our multi-tree structure, but without requiring a huge amount of memory (due to the sharing of parts), and without taking too much time (fusion and archetype extraction are preformed locally and not globally).

Part IV

Cost-sensitive Learning

Chapter 11

Minimising Costs in Rule-Based Classifiers

In this chapter we address the issue of constructing models that mimimise misclassification and test costs when these costs are known or unknown in training time. Therefore, the objective is to get models that minimise the total cost of errors instead of minimise the overall number of errors.

For the purpose of learning models that would behave well in a wide variety of contexts, we employ ROC analysis, concretely, we introduce a new and effective method to compute the AUC value. This new method can also be employed as a splitting criterion for constructing decision trees that behave well for any cost context. We also address this issue for multi-classes problems. Some of the contents of this chapter have appeared in [Flach *et al.*, 2003; Ferri *et al.*, 2002a; Estruch *et al.*, 2002a].

We also address the problem of learning models that minimise the cost of testing attributes when predicting new cases. We present some methods to learn models which reduce this cost without reducing their accuracy significatibaly. These methods have been published in [Estruch *et al.*, 2002d]. The techniques will be illustrated for decision-tree learning, but many of them are directly extensible to other kinds of models that produce disjoint rules, such as ILP, IFLP and rule learning.

11.1 Introduction

Traditionally, classification accuracy (or error), i.e., the percentage of instances that are correctly classified (respectively incorrectly classified) has been used as a measure of the quality of classifiers. However, in many situations, not every misclassification has the same consequences, and problem-dependent misclassification costs have to be taken into account. If the cost parameters are not known at training time, Receiver Operating Characteristic (ROC) analysis (see Section 5.2.2) can be applied [Provost and Fawcett, 1997; Swets *et al.*, 2000]. ROC analysis provides tools to distinguish classifiers that are optimal under some class and cost distributions from classifiers that are always sub-optimal, and to select the optimal classifiers once the cost parameters are known.

This chapter is concerned with taking costs into account when learning decision trees. If costs are known at training time, the training algorithm could be made cost-sensitive, e.g. by incorporating costs in the splitting criterion, or by modifying the distribution of the classes in the training dataset (see Section 5.2.3). However, it has been shown that such cost-sensitive techniques do not lead to trees with lower costs [Drummond and Holte, 2000; Elkan, 2001] and that cost-sensitive class labelling is more effective [Bradford *et al.*, 1998].

In this chapter we assume that costs are unknown at training time. Clearly, each of the 2^n possible labellings of the n leaves of a given decision tree establishes a classifier, and we can use ROC analysis to determine the optimal labellings among them. However, this set of classifiers has special properties (e.g., for any classifier there is another one making opposite predictions) which allows us a more direct computation of the optimal labellings. We prove that there are $n+1$ of these, which are determined by a simple ordering on the leaves of the tree.

Thus, from a cost-sensitive perspective it makes sense to view a decision tree as an unlabelled tree with an ordering on the leaves. Furthermore, this suggests to use the area under the ROC curve (AUC), obtained by plotting the $n+1$ optimal labellings in the ROC space, to evaluate the quality of a decision tree (or any other partitioning of instance space). A natural question is then whether existing decision tree algorithms – which aim at optimising the accuracy of a single labelling – also lead to good AUC values, or whether we can do better by adapting the algorithm. Based on this idea, we define a simple AUC-based splitting criterion.

We also address the same problem for multi-class datasets, by introducing some other new splitting criteria that lead to trees with better AUC, without sacrificing accuracy if a single labelling is chosen.

Finally, a proposal for learning decision-trees that minimise the test cost of predicting new cases is presented. In particular, we show a modification of classical splitting criteria that tends to produce decision trees with lower test cost. With regard to the multi-tree structure, we adapt the archetype technique in order to extract accurate and comprehensible models that also minimise the test cost.

11.2 Finding Optimal Labellings of a Decision Tree

A decision tree classifier can be represented by a point in the ROC space. However, if we change the class assignment of one leaf, we obtain a different classifier and hence a different point in the ROC space. Note that this change can be made a posteriori, after the tree was learnt or pruned. By changing in many different ways the assignments of each leaf of the tree we can obtain different trees. In what follows we will call *labelling* a set of assignments to each tree leaf.

The idea is to view the ROC curve of a decision tree not as the three-point curve given by a single labelling together with the two default classifiers, but as the convex hull defined by all the possible labellings. The problem is that given n leaves and c classes, there are c^n possible labellings. Although this value alone can make this intractable for many trees even for two classes, the problem gets worse if we consider that we would need to compute the convex hull of these c^n points. Note that the cost of computing the convex hull of N points in a d-dimensional space is in $O(N \log N + N^{d/2})$ [J.D. Boissonat, 1998]. Consequently, one relevant question is whether there is a way to restrict these c^n combinations and obtain the same ROC curve.

11.2.1 Notation

Given a set of tree leaves l_k ($1 \leq k \leq n$) and a training set S with c possible classes ($1 \leq i \leq c$), we denote by E_k the number of examples of S that fall under leaf l_k, and we denote by E_k^i the number of examples of S that fall under leaf l_k of class i. The k subscript in E_k^i can be dropped when the leaf is clear from context. A *labelling* is defined as a set of pairs of the form (k, i), where k represents the leaf l_k and i represents the class assigned to that leaf. The set of all possible labellings is denoted by Λ. Clearly, the cardinality of Λ is c^n.

In what follows, we study for 2-class problems how we can restrict the 2^n labellings but still obtain the points on the convex hull. We will denote the two classes: $+$ and $-$. We also assume the following properties:

$$E_k^+ \geq 0, E_k^- \geq 0, E_k^+ + E_k^- > 0, \forall\, 1 \leq k \leq n,$$

$$\sum_{1 \leq k \leq n} E_k^- > 0, \texttt{and} \sum_{1 \leq k \leq n} E_k^+ > 0$$

That means that there are no empty leaves and that there exists at least one example of each class.

We use the following notation for cost matrices $C_{i,j}$:

		Actual	
		+	−
Predicted	+	C_{++}	C_{+-}
	−	C_{-+}	C_{--}

where all the costs are greater or equal than 0. Additionally, $C_{-+} > C_{++}$, $C_{+-} > C_{--}$, since misclassification costs must be higher than the hits. Given a leaf l_k, we define $Cost_k^i$ as the cost of the examples under that leaf if class i would be assigned:

$$Cost_k^i = \sum_j E_k^j \cdot C_{i,j}$$

The best assignment for a leaf l_k is then defined as:

$$Best_k = \arg\min_i Cost_k^i$$

The optimal labelling S_{opt} for a given cost matrix C is then given by:

$$S_{opt} = \{(k, Best_k)\}_{1 \leq k \leq n}$$

which means that each leaf is assigned the class that minimises the cost for the cost matrix C.

11.2.2 Subset of Labellings forming the Convex Hull

In this section we determine the subset of decision tree labellings on the convex hull.

Lemma 12 *Given a leaf of a decision tree for a 2-class problem with the distribution E^+ and E^-, and given a cost matrix C, the cost is minimised if the leaf is assigned class "+" when*

$$\frac{E^+}{E^- + E^+} \geq \frac{(C_{+-} - C_{--})}{(C_{-+} - C_{++}) + (C_{+-} - C_{--})}$$

and assigned class "-" otherwise.

Proof. The cost of this leaf will be assigned to + iff $Cost_+ \leq Cost_-$, i.e.

$$E^+ \cdot C_{++} + E^- \cdot C_{+-} \leq E^+ \cdot C_{-+} + E^- \cdot C_{--}$$

$$\frac{E_+}{E_- + E_+} \geq \frac{(C_{+-} - C_{--})}{(C_{-+} - C_{++}) + (C_{+-} - C_{--})}$$

□

The value on the left hand side is defined as the *local positive accuracy* of a leaf l_k, and is denoted by r_k. This result has also been used elsewhere to assign classes (see e.g. [Elkan, 2001]), but we will use it to *order* the leaves. The value on the right hand side of the equation is called the *cost ratio* (CR). In particular, when r_k = CR either class can be assigned arbitrarily.

The main definition of this section is the following.

Definition 13 (Optimal labellings) *Given a decision tree for a problem with 2 classes formed by n leaves* $\{l_1, l_2, \ldots, l_n\}$ *ordered by local positive accuracy, i.e,* $r_1 \geq r_2, \ldots, r_{n-1} \geq r_n$, *we define the set of optimal labellings* $\Gamma = \{S_0, S_1, \ldots, S_n\}$ *where each labelling* S_i $(0 \leq i \leq n)$ *is defined as* $S_i = \{A_i^1, A_i^2, \ldots, A_i^n\}$ *where* $A_i^j =$ *(j,+) if* $j \leq i$ *and* A_i^j = *(j,-) if* $j > i$.

The following three lemmas are needed to establish the main result of this section.

Lemma 14 *Given a decision tree for a problem with 2 classes with n leaves, the labelling that minimises the cost according to the training set and an arbitrary cost matrix belongs to the set of optimal labellings* Γ.

Proof. The cost matrix has one degree of freedom expressed with the CR. Imagine that the CR is 1, then all the leaves will be set to −, according to Lemma 12 (in the case r_k = CR we also select −). This labelling is in Γ. This solution minimises the cost of any matrix until $r_1 \leq$CR$\leq r_2$. Then, leaf l_1 will change its assignment to + according to Lemma 12; this labelling also belongs to Γ. We can repeat this argument until CR=0, where all the leaves will be set to +. Thus, there are n+1 states that correspond to the labellings in Γ. □

[Blockeel and Struyf, 2001] used the same set of assignments. However, no theoretical properties were discussed. Lemma 15 shows that the set of optimal labellings is sufficient for the calculation of the convex hull. We now proceed to show that these points are also necessary.

Lemma 15 *Given three labellings from the set of optimal labellings* Γ*:* S_{i-1}, S_i, S_{i+1} $(1 \leq i \leq n-1)$*, the point in the ROC space corresponding to classifier* S_i *is above the convex hull formed by (0,0), (1,1), and the points in the ROC space corresponding to classifiers* S_{i-1} *and* S_{i+1} *if and only if*

$$\frac{E_i^+}{E_i^- + E_i^+} > \frac{E_{i+1}^+}{E_{i+1}^- + E_{i+1}^+},$$

Proof. The three points in the ROC space corresponding to S_{i-1}, S_i, S_{i+1} are:

$$P_{i-1} = \left(\frac{x_{i-1}}{x_t}, \frac{y_{i-1}}{y_t}\right),$$

$$P_i = \left(\frac{x_{i-1} + E_i^-}{x_t}, \frac{y_{i-1} + E_i^+}{y_t}\right),$$

$$P_{i+1} = \left(\frac{x_{i-1} + E_i^- + E_{i+1}^-}{x_t}, \frac{y_{i-1} + E_i^+ + E_{i+1}^+}{y_t}\right)$$

where

$$x_t = \sum_{1\leq j\leq n} E_j^- \quad y_t = \sum_{1\leq j\leq n} E_j^+$$

$$x_{i-1} = \sum_{1\leq j\leq i-1} E_j^- \quad y_{i-1} = \sum_{1\leq j\leq i-1} E_j^+.$$

Obviously,

$$\frac{x_{i-1}}{x_t} \leq \frac{x_{i-1} + E_i^-}{x_t} \leq \frac{x_{i-1} + E_i^- + E_{i+1}^-}{x_t}.$$

Thus, according to the definition of ROC curve, we only want to know when P_i is *above* the straight line that joins P_{i-1} and P_{i+1}, focusing on the y coordinate.

The formula of a straight line that joins two points $P_1{=}(X_1{,}Y_1)$ and $P_2{=}(X_2{,}Y_2)$ is:

$$y = \frac{Y_2 - Y_1}{X_2 - X_1}(x - X_1) + Y_1$$

Substituting $P_1 = P_{i-1}$ and $P_2 = P_{i+1}$, the y coordinate of P_i will be above iff:

$$\frac{y_{i-1} + E_i^+}{y_t} > \frac{\frac{y_{i-1}+E_i^+ + E_{i+1}^+}{y_t} - \frac{y_{i-1}}{y_t}}{\frac{x_{i-1}+E_i^- + E_{i+1}^-}{x_t} - \frac{x_{i-1}}{x_t}} \cdot \left(\frac{x_{i-1} + E_i^-}{x_t} - \frac{x_{i-1}}{x_t}\right) + \frac{y_{i-1}}{y_t}$$

$$iff\, \frac{y_{i-1}+E_i^+}{y_t} > \frac{\frac{E_i^+ + E_{i+1}^+}{y_t}}{\frac{E_i^- + E_{i+1}^-}{x_t}} \cdot \left(\frac{E_i^-}{x_t}\right) + \frac{y_{i-1}}{y_t}$$

$$\text{iff } E_i^+ \cdot E_{i+1}^- > E_i^- \cdot E_{i+1}^+,$$

$$\text{iff } \frac{E_i^+}{E_i^- + E_i^+} > \frac{E_{i+1}^+}{E_{i+1}^- + E_{i+1}^+}$$

□

We have shown this result for three consecutive classifiers of the set of optimal labellings; however, it also holds for three non-consecutive classifiers.

Lemma 16 *Given three labellings from the set of optimal labellings* Γ*:* S_{i-1}, S_i, S_{i+1} *(*$1 \leq i \leq n-1$*) such that* $r_i = r_{i+1}$*, it is not necessary to consider the point in the ROC space corresponding to* S_i*, because it will not affect the convex hull.*

Proof. : If $r_i = r_{i+1}$ then

$$\frac{E_i^+}{E_i^- + E_i^+} = \frac{E_{i+1}^+}{E_{i+1}^- + E_{i+1}^+},$$

which, according to Lemma 15, means that the point in the ROC space corresponding to S_i is placed just on the straight line between the points in the ROC space corresponding to S_{i-1} and S_{i+1}. □

We can now formulate the main result of this section.

Theorem 17 *Given a decision tree for a problem of 2 classes with* n *leaves, the convex hull of the* 2^n *possible labellings is formed by exactly those ROC points corresponding to the set of optimal labellings* Γ*, removing repeated leaves with the same local positive accuracy.*

Proof. From Lemma 14 we can easily derive that all the ROC points that are on the convex hull from the 2^n possible labellings belong to the ROC points generated from the set of optimal classifiers. We only have to show that all the ROC points from the set of optimal labellings are on the convex hull. Suppose we have three consecutive labellings S_{i-1}, S_i, S_{i+1} where S_{i-1} and S_{i+1} are on the convex hull. Lemma 15 has shown that S_i will be above the convex hull iff $r_i \geq r_{i+1}$, which is the case since the set of optimal labellings is ordered by local positive accuracy. In the case that $r_i = r_{i+1}$ we have, from Lemma 16, that we can remove one of them. □

The relevance of Theorem 17 is that computation of the convex hull of the 2^n possible labellings of the n leaves of a decision tree is equivalent to ordering the leaves by local positive accuracy.

Example 17 *Suppose we have a decision tree with three leaves and the following training set distribution:*

	$+$	$-$
$Leaf\ 1$	3	5
$Leaf\ 2$	5	1
$Leaf\ 3$	4	2

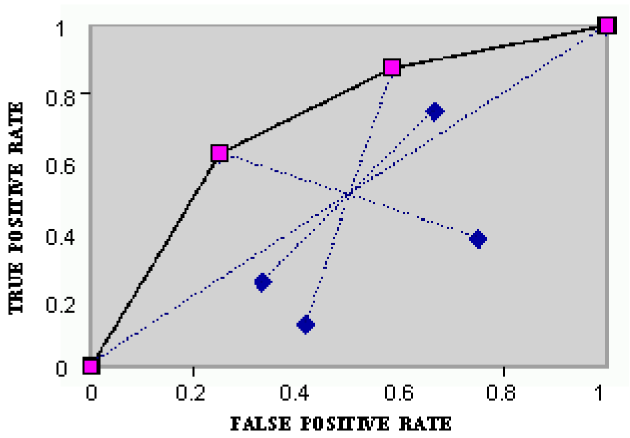


Figure 11.1: ROC points and convex hull of set Λ

There are 2^3=8 possible classifiers corresponding to each labelling in Λ. Figure 11.1 represents the ROC points of these classifiers. As can be seen in the figure, the points are mirrored through the point (0.5, 0.5), because for each labelling there is another labelling assigning the opposite class to each leaf.

We first order the leaves by the local positive accuracy and then we generate the set of n+1= 4 optimal labellings:

	$+$	$-$	S_0	S_1	S_2	S_3
Leaf 1	5	1	-	+	+	+
Leaf 2	4	2	-	-	+	+
Leaf 3	3	5	-	-	-	+

If we plot the ROC points of these 4 combinations, these are the points corresponding to the convex hull of Λ, which are shown in Figure 11.1 as squares.

A final question is what to do with empty leaves, a case that we have not considered in the previous results because we excluded this case in the assumptions. Empty leaves can be generated when there are splits with more than two leaves, some of which may not cover any example, i.e., E^+=0 and E^-=0. One easy solution to this problem is to use some kind of smoothing (such as Laplace or m-estimate) for E^+ and E^-. Another option is to assign a local positive accuracy 0.5 and work with the leaf without cardinality, not affecting the ROC curve.

11.3 AUC-based Decision Tree Evaluation and Construction

The previous analysis supports the interpretation of a decision tree as having several possible labellings of the leaves, or alternatively, an ordering on the leaves. We propose to use the area under the ROC curve (AUC) obtained from these labellings to evaluate the quality of the tree. Notice that if the accuracy of the tree is 100%, all leaves are pure and the ROC curve covers the whole space. If the tree consists of a single unsplit leaf, the two labellings of this leaf correspond to the two default classifiers and the area under the curve is 0.5. Also notice that we can even calculate the AUC of a single labelling, i.e. the area under the curve $(0,0) - (FPR,TPR) - (1,1)$, which is $(TPR - FPR + 1)/2$, i.e. the average of positive and negative accuracies.

11.3.1 The AUC Metric for Decision Tree Evaluation

In order to compute the area under the ROC curve we employ the leaf ordering from the previous section to compute the areas of each trapezoid. Specifically, it is easy to compute the area between two consecutive points P_{i-1} and P_i in the ROC curve given by the set Γ:

$$A(P_{i-1}, P_i) = \frac{E_i^-}{x_i} \frac{2y_{i-1} + E_i^+}{2y_i}$$

where y_{i-1}, x_t and y_t are as defined in Lemma 15. Since the first point is $P_0 = (0,0)$, we can define AUC as follows.

Definition 18 (AUC) *Let Γ be the set of optimal labellings of a decision tree with n leaves, then the AUC metric is defined as*

$$AUC(\Gamma) = \sum_{i=1..n} A(P_{i-1}^j, P_i^j) = \sum_{i=1..n} \frac{E_i^-}{x_i} \frac{2y_{i-1} + E_i^+}{2y_i}$$

$$= \frac{1}{2x_i y_i} \sum_{i=1..n} E_i^- \left[\left(\sum_{j=1..i-1} 2E_j^+ \right) + E_i^+ \right]$$

(See Lemma 15 for the meaning of the symbols.)

AUC is like any other machine learning metric in that it is a population statistic which needs to be estimated from a sample. We can use the standard techniques of using a test set or cross-validation to obtain such an estimate. In the case of a test set, note that the leaf ordering is obtained during training, while the leaves' positive and negative coverage is determined on the test set. Consequently, the ROC curve on the test set may not be convex (it is, however, monotonically non-decreasing by construction). Definition 18 is a general geometric construction which does not assume convexity of the curve.

11.3.2 The AUCsplit Splitting Criterion for Decision Tree Construction

In Chapter 5 we have argued that AUC is a better metric than accuracy for evaluating decision trees when class and cost distributions are unknown at training time. However, the existing methods for growing decision trees typically use splitting criteria based on error/accuracy or discrimination. In this subsection, we propose an AUC-based splitting criterion.

Without the results we have just introduced in Section 11.2, computing the AUC corresponding to a set of n leaves would be computationally expensive, especially if splits have more than two children. Using the optimal labelling set Γ, the AUC of the leaves under a split can be computed efficiently. In particular, given several possible splits for growing the tree, where each split consists of a set of new leaves, we can compute the ordering of these leaves and calculate the corresponding ROC curve. The area under this curve could be compared to the areas of other splits in order to select the best split. More precisely, we can use the previous formula for AUC(Γ). This yields a new splitting criterion.

Definition 19 (AUCsplit) . *Given several splits s_j, each one formed by n_j leaves $\{l^j_1, l^j_2, ..., l^j_{n_j}\}$, then the best split is the one that maximises:*

$$AUCsplit(s_j) = \sum_{i=1..n_j} A(P^j_{i-1}, P^j_i)$$

where the points P^j_i are obtained in the usual way (sorting the leaves of each split by local positive accuracy).

The first question that arises with a new splitting criterion is how it differs from other criteria previously proposed. To answer this question, let us review the general formula of other well-known splitting criteria (see Section 2.2.1). In general, these splitting criteria find the split with the lowest $I(s)$, where $I(s)$ is defined as:

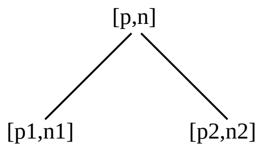
$$I(s) = \sum_{j=1..n_j} p_j \cdot f(p^+_j, p^-_j)$$

where n_j is the number of alternatives in the split, and p_j is the probability of falling into that node in the split (cardinality of child node divided by the cardinality of the parent node). Using this general formula, each splitting criterion implements a different function $f(a, b)$.

Consider for instance the following two splits:

The children have the same weighted average impurity in both cases. In order to see that the left is a better split than the right (assuming $a \neq$b), we need to take the impurity of the parent into account. In contrast, AUCsplit evaluates the quality of the whole split (parent + children) and cannot be reduced to a difference in impurity between parent and children. The left split has AUCsplit=$a/(a+b)$ (assuming $a > b$), while the right split has AUCsplit=0.5, indicating that nothing has been gained in ROC space with respect to the default diagonal from (0,0) to (1,1).

An interesting relationship can be established with the Gini index. Consider the following binary split:



If the left child has higher local positive accuracy, then we have:

$$AUCsplit = \frac{1}{2}\left(\frac{p_1}{p} - \frac{n_1}{n} + 1\right) = \frac{p_1 n - p n_1 + pn}{2pn} = \frac{p_1 n + p n_2}{2pn}$$

It is interesting to note that the denominator of this expression is the Gini index of the parent, and the enumerator could be called a mutual Gini index of the children given the parent.

Finally, we have to consider the computational complexity of calculating the AUC-split with respect to other well-known splitting criteria. Let n denote the maximum number of children in all the splits. Then, if we have k partitions, the selection of the best split by using any of the information measures $I(s_j)$ requires, for each partition, n computations of the entropy formula, that can be considered in $\mathcal{O}(1)$. Consequently, the cost would be in $\mathcal{O}(k \cdot n)$. On the other hand, the selection of the best split by using the AUCsplit(s_i) requires n·log$n$ for sorting the n nodes, and n computations of the A(·,·) formula that can be considered in $\mathcal{O}(1)$. Consequently, the cost would be $k \cdot (n\cdot\log\, n + n)$, which is in $\mathcal{O}(k \cdot n\cdot\log\, n)$. This difference in log n is negligible especially if we realise that the number of children of a partition is 2 for numerical attributes and very small for nominal attributes.

11.4 Extending AUC Measure and Splitting Criterion

The previous section has been devoted to analyse the problem for two classes. In the rest of this chapter we are going to concentrate on multi-class problems, i.e., classification problems with more than two classes.

Theoretically, it is possible to compute the convex hull of a set of multi-dimensional points and then try to obtain the volume under the convex hull, in particular, a polytope [J.D. Boissonat, 1998]. However, the best algorithm for the convex hull generation is in $\mathcal{O}(N \log N + N^{d/2})$ [J.D. Boissonat, 1998; Lane, 2000], where N is the number of points and d is the dimensionality, which is defined as $c\times$ $(c-1)$ where c is the number of classes.

Since this would be unfeasible in practice, we have to find an approximation for the AUC measure for more than two classes.

11.4.1 *M* Function

[Hand and Till, 2001] have presented an extension based on the idea that the AUC measure tells "how well separated are the estimated distributions of $p(x)$ for class 0 and class 1". If we have more than two classes we *can* compute the separateness of each pair of classes and then average all the values. This simple idea is just called the M function [Hand and Till, 2001]. Let us describe how this function can be computed.

First of all, all the examples (either positive or negative) are ranked in increasing order taking into account the estimated probability of belonging to class 0. They define S_0 as the sum of the ranks of the class 0 test points.

Then, the AUC for two classes 0 and 1 is defined as:

$$\hat{A} = \frac{S_0 - n_0(n_0+1)/2}{n_0 n_1}$$

Note that this seems a different way to compute the AUC, but it is equivalent to our method proposed in the previous sections. Let us see it with an example:

Example 18 *Given the following tree and training distribution*[1]*:*

	$+$	$-$
$Leaf$ 1	5	1
$Leaf$ 2	4	2
$Leaf$ 3	3	5

[1]The class $-$ is considered as the class 0 when computing the AUC value with the Hand and Till's formula.

since there are no ties there is no need to use smoothing and the order in positive probability is 1,2,3.

If we have the following test set distribution and labellings:

	$+$	$-$	S_0	S_1	S_2	S_3
Leaf 1	4	3	-	$+$	$+$	$+$
Leaf 2	6	3	-	-	$+$	$+$
Leaf 3	5	4	-	-	-	$+$

following Hand and Till's formula, we derive the estimated probability values of belonging to class $-$ for test examples, positives f_i, and negatives g_i: $f_1 ... f_5$ and $g_1 ... g_4$ are derived from the probability given by the third leaf. Consequently they are ranked with values from 1 to 9 (with mean 5). $f_6..f_{11}$ and $g_5..g_7$ fall in leaf 2 and consequently they are ranked with values from 10 to 18 (with mean 14). Finally $f_{12}..f_{15}$ and $g_8..g_{10}$ fall in leaf 1 and consequently they are equally ranked with values 19 to 25 (with mean 22).

Consequently, (using the mean of the ranks) for each group we have:

$$S_0 = 5 \cdot 5 + 6 \cdot 14 + 4 \cdot 22 = 197$$

and

$$\hat{A} = \frac{197 - 15(15+1)/2}{150} = \frac{197 - 120}{150} = 0.513$$

We have that AUC (by using our method) is also 0.513.

Seen this example, if we consider $\hat{A}(i,j)$ as the AUC with classes i, j, then the generalised M function is:

$$M = \frac{2}{c(c-1)} \sum_{i<j} \hat{A}(i,j)$$

We will call this M function measure *MAUC*, i.e. Multi-class Area Under the ROC Curve extension. Clearly, MAUC and AUC are equivalent when c=2.

11.4.2 MAUC Splitting Criterion

The first problem is that this formulation of the M function is based on a ranking of examples. Consequently it has cost $\mathcal{O}(m \cdot \log(m))$, where m is the number of examples. If this has to be done for the c classes, it can be intractable. This high computational complexity would become an important obstacle for using it as splitting criterion.

Fortunately, the complexity could be reduced if we know that for all the examples under the same node the rank would be the same. Consequently, we only have to rank the nodes, as we made in the two-classes case, and the complexity of the ordering would be O(n·log$n$) where n is the number of nodes. Given a set of nodes S, let us define the area just considering two classes a and b:

$$AUC_{ab}(S) = \sum_{i=1..|s|} A_{ab}(P^{ab}_{i-1}, P^{ab}_{i})$$

where the points P^{ab}_i are obtained in the usual way (sorting the leaves of each split by local positive accuracy, just taking into account classes a and b).

And now, we can redefine the M function as MAUC in the following way:

$$MAUC(S) = \frac{2}{c(c-1)} \sum_{a<b} AUC_{ab}(S)$$

It is easy to see that for all the combinations of classes needed for computing the MAUC measure we would have $\mathcal{O}(c^2 \cdot n \cdot \log(n))$.

Finally, we can define the splitting criterion in the following way:

Definition 20 (MAUCsplit) *Given several splits s_j, each one formed by n_j leaves $\{l^j_1, l^j_2, ..., l^j_{n_j}\}$, then the best split is the one that maximises MAUC(s_j).*

11.4.3 Using Smoothing in the AUC and MAUC measures

Before evaluating this new splitting criterion, we have to discuss some peculiar things about the ordering that turns to be more important in multi-class problems.

Given two nodes with training distribution [3,0] and [7,0] for two classes 0 and 1, both of them have the same local positive accuracy 1. If the ordering of nodes is not performed carefully, the result is that the node [3,0] could come before [7,0] in the ordering. This is a mistake, since the node [3,0] has more probability of making bad guesses than the node [7,0] because the latter has more cases that reinforce the probability.

An possible solution for this is the use of a simple smoothing, such as the Laplace correction. With this, the ordering is based on the smoothed local positive accuracy, which for the previous examples would be [4,1] and [8,1]. Now the latter node has greater smoothed local positive accuracy.

It is also easy to show that this only affects nodes with equal local positive accuracy. If a node a has strictly lower local positive accuracy than a node b, then a has strictly lower smoothed local positive accuracy than b.

Another important issue about the AUC measure is when the test set does not have examples of one class. In this case, the AUC has no sense. This case is more frequent when the number of classes is high.

In what follows, all the experiments perform AUC measures with Laplace correction (just for making the ordering from the training set) and the partitions we have made (in k-fold cross-validation for instance) always lead to test sets with examples of all classes (test sets with no examples of one class are avoided).

11.5 New Splitting Criteria for Multi-Class Problems

The AUC measure tells "how well separated are the estimated distributions of $p(x)$ for class 0 and class 1" [Hand and Till, 2001]. Why do not we develop other measures that try to approximate how well separated two distributions are?

11.5.1 New Measures

One measure of this kind is the well-known Mean Squared Error measure:

$$MSE = \frac{\sum_{i=1..m} \sum_{j=1..c} (f(i,j) - p(i,j))^2}{m \cdot c}$$

where $f(i,j)$ is the actual probability of example i to be of class j and $p(i,j)$ is the estimated probability of example i to be of class j. The denominator gives a normalised MSE between 0 and 1. For classification problems, $f(i,j)$ will be always 0 or 1, depending on the class.

Another measure is the log-loss, which is claimed to be a measure of the goodness of probability estimates [Bernardo and Smith, 1994; Mitchell, 1997]:

$$LogLoss = \frac{- \sum_{i=1..m} \sum_{j=1..c} (f(i,j) \log_2 p(i,j))}{m}$$

Note that these two measures are closely related to MAUC. As we have discussed, the MAUC measure is similar of what is measured by the expression $(f(i,j) - p(i,j))^2$ or by the expression $(f(i,j) log_2 p(i,j))$.

These two measures have the advantage that are much easier to be understood and computed. Obviously, they have no problems of generalisation for more than 2 classes.

Another measure is the macro-average accuracy which is the mean of the partial class accuracies. This is equivalent to the AUC of a single labelling. This last measure is an over-simplification. We have essayed it with bad results. Thus, we will ignore this one and will study some experiments with the previous ones: MSE and LogLoss.

11.5.2 Adaptation as Splitting Criteria

Consider a partition with n nodes. Since for classification problems, $f(i,j)$ will be always 0 or 1, depending on the class, we have the following equation:

$$MSEsplit = \frac{\sum_{i=1..n} \sum_{j=1..c} card(i,j) \cdot \sum_{k=1..c} if(k=j)(1-p(i,j))^2 else(0-p(i,j))^2}{n \cdot c}$$

where $card(i,j)$ is the number of examples of class j in the node i, and $p(i,j)$ is the estimated probability of class j for the node i.

The MSEsplit that this has cost $\mathcal{O}(n \cdot c^2)$ where the initial MSE had $\mathcal{O}(m \cdot c)$. Note that for a problem with 100.000 examples and 5 classes, and given a first split at the top of the tree with three children, we would have a value of $\mathcal{O}(n \cdot c^2)$=75 for the first expression and a value of $\mathcal{O}(m \cdot c)$=500,000 for the second.

The formula for logloss is simpler, because many cases are just removed when $f(i,j)$=0. Consequently, we have:

$$LogLossSplit = \frac{\sum\limits_{i=1..n} \sum\limits_{j=1..c} card(i,j) \cdot (f(i,j) \log_2 p(i,j))}{n}$$

The cost is just $\mathcal{O}(n \cdot c)$ Finally, it is important to note that, in the same way that expected error is not a very good splitting criterion for obtaining low global errors, it is possible that MAUCsplit is not the best formulation for minimising MAUC, as well as MSEsplit could not be the best formulation for minimising MSE and LogLossSplit could not be not the best formulation for minimising LogLoss.

11.6 Experimental Evaluation

We evaluate the previous methods by using 28 datasets extracted from the UCI repository. Details of the datasets employed can be found in Appendix A.

We compare the splitting criteria introduced in this chapter (M-AUCsplit, LogLoss, and MSE) with Gain Ratio (only considering splits with at least average gain as is done in C4.5 [Quinlan, 1993]), mgini (as used in CART [Breiman *et al.*, 1984]) and DKM [Kearns and Mansour, 1996]. All the experiments have been done within the SMILES system (see Appendix C) that includes all of these criteria, the labelling method and the AUC computation. The use of the same system for all the methods makes the criteria comparison more impartial because all the other things remain equal. The experiments were performed without pruning. We have also used frequency smoothing (Laplace correction) for the nodes in each split, because it is favourable for the AUC measure for all methods. This same conclusion is shown in [Provost and Domingos, 2002]. Accuracy of Gain Ratio results are slightly worse when smoothing is used, although AUC values are still better. In order to make comparisons significant, we have repeated 10-fold cross validation 10 times, making a total of 100 learning runs for each pair of dataset and method.

The first thing to be considered is the behaviour of the splitting criteria with the accuracy evaluation measure. Table 11.1 shows the accuracy results. Apart from the DKM criterion, the performance is similar. The best accuracy is obtained by the new LogLoss Criterion. Gain Ratio and MSE also get positive results.

#	Auch	C45	DKM	LogLoss	Mgini	MSE
1	75.48	77.18	78.18	79.45	76.79	78.44
2	94.22	93.48	93.96	94.19	93.75	93.04
3	92.86	92.25	81.66	93.54	91.02	93.14
4	89.02	88.95	62.12	90.44	88.90	89.28
5	99.37	99.40	81.34	99.34	99.39	99.60
6	46.95	48.02	47.11	48.46	47.48	47.72
7	84.69	89.86	58.08	93.69	92.31	89.97
8	37.46	34.54	35.77	36.85	34.85	38.54
9	64.80	66.93	62.00	67.37	64.67	65.80
10	72.75	71.37	69.88	72.12	71.94	71.81
11	48.39	50.27	52.36	50.88	50.76	50.33
12	74.47	78.13	69.20	71.93	74.53	71.67
13	61.81	65.42	61.42	63.89	64.64	63.53
14	78.58	78.33	71.72	81.00	77.64	77.25
15	94.02	93.47	92.51	95.26	93.63	94.65
16	94.87	94.47	70.93	93.47	94.20	94.60
17	63.35	62.53	56.47	63.44	61.82	64.06
18	93.67	91.91	97.98	93.51	91.02	99.44
19	74.68	74.47	68.77	67.17	72.98	66.78
20	97.47	97.62	96.20	98.65	97.73	97.51
21	95.38	92.10	81.48	93.76	92.19	92.38
22	67.96	69.25	66.68	71.21	68.51	68.67
23	56.63	60.12	53.25	61.25	59.38	60.38
24	96.33	95.80	48.51	95.82	95.72	96.16
25	61.47	62.53	64.33	60.87	62.60	62.13
26	98.85	98.91	94.48	99.36	98.73	99.14
27	77.72	79.42	69.00	77.43	79.41	76.27
28	91.47	92.59	60.82	91.94	92.53	92.35
GeoMean	75.77	76.33	67.64	76.59	75.94	76.21

Table 11.1: Accuracy depending on the splitting criterion.

#	Auch	C45	DKM	LogLoss	Mgini	MSE
1	0.815	0.825	0.760	0.836	0.819	0.835
2	0.981	0.973	0.985	0.973	0.977	0.976
3	0.972	0.967	0.848	0.973	0.920	0.973
4	0.956	0.955	0.650	0.960	0.954	0.957
5	0.999	1.000	0.904	0.999	0.999	1.000
6	0.649	0.638	0.631	0.670	0.627	0.658
7	0.961	0.988	0.813	0.992	0.992	0.988
8	0.548	0.547	0.605	0.573	0.539	0.602
9	0.635	0.663	0.491	0.648	0.621	0.651
10	0.892	0.891	0.903	0.887	0.884	0.876
11	0.650	0.655	0.653	0.668	0.659	0.654
12	0.776	0.747	0.422	0.715	0.684	0.723
13	0.692	0.697	0.652	0.742	0.721	0.720
14	0.828	0.817	0.772	0.834	0.818	0.818
15	0.983	0.981	0.978	0.981	0.985	0.984
16	0.988	0.981	0.852	0.977	0.982	0.989
17	0.669	0.653	0.563	0.663	0.660	0.659
18	0.981	0.972	0.992	0.975	0.969	1.000
19	0.758	0.758	0.683	0.653	0.736	0.634
20	0.992	0.993	0.968	0.991	0.992	0.990
21	0.984	0.970	0.808	0.967	0.974	0.975
22	0.738	0.760	0.671	0.773	0.685	0.763
23	0.371	0.430	0.367	0.421	0.378	0.526
24	0.997	0.996	0.788	0.996	0.995	0.997
25	0.775	0.775	0.772	0.772	0.771	0.769
26	0.990	0.993	0.718	0.995	0.984	0.993
27	0.878	0.882	0.729	0.863	0.882	0.856
28	0.971	0.977	0.757	0.970	0.969	0.977
GeoMean	0.817	0.821	0.721	0.821	0.808	0.826

Table 11.2: AUC depending on the splitting criterion.

Secondly, we evaluate the behaviour of these splitting criteria with respect to the AUC value. Table 11.2 shows the AUC of the classifiers generated with the considered splitting criteria. Again, DKM obtains the worst results, and MSE the best behaviour. LogLoss, Gain Ratio and Auch have a similar performance.

To conclude, the experimental evaluation has shown the utility of the splitting criteria defined in this chapter. The new criteria (Auch, logloss and MSE) obtain better results in accuracy and AUC than some classic splitting criteria such as DKM or mgini. With respect to the popular Gain Ratio criterion, the results in terms of accuracy are similar, or even slightly better with Logloss. Regarding the AUC measure, again the results are identical, except from MSE that improves in a few percentage GainRatio.

11.7 Test Cost Minimisation

There is another kind of costs associated to the learning process that can be extremely important in some applications. This is the cost of performing the tests that a model requires to make a prediction. This kind of cost is often called test cost. Section 5.2.4 has presented the main issues on this kind of costs.

In this part of the dissertation, we study some mechanisms to learn models that are able to make predictions with a high level of hits and minimise the test cost to perform such predictions.

For this work, we are going to implement the following generalisation of previous approaches (see Section 5.2.4). Consider the optimality Opt obtained by any splitting criterion (Gain Ratio, Gini, AUCsplit, MSEsplit, etc.). We use the following formula to modify Opt:

$$Opt'(S, A) = \frac{Opt(S, A)}{(Cost(A) + 1)^w}$$

where A is an attribute, S is the set of examples where the split is tested, $Cost(A)$ is the cost associated to attribute A, and w is a constant that determines the relative importance of cost versus information gain. This weight w tells how much the test cost affects the splitting criteria. Note that the relevance of test cost can also be modified quite easily by augmenting the absolute value of the test cost vector.

To obtain $Cost(A)$, it is important to note that numerical attributes can be used more than once in the same branch. Consider e.g. a condition $X < 3.2$; this could be followed by an additional partition below on the same attribute in the same branch such as $X < 1.3$ and $X \geq 1.3$. It is not sensible to compute the test cost of attribute X more than once. For this reason, it is more accurate to take into account this cost once. This gives an exact measurement of test cost.

This modification provides a way to construct trees and multi-trees that have lower average test costs per example. Obviously, if test cost is given too much importance, then AUC and accuracy can be affected, as we will see in the experiments.

Despite the effect and good results of the previous approach, it is not much wise to use a test cost sensitive criterion and then select the best tree using non-test cost sensitive criteria. Therefore, we have studied a method to extract a single solution from the multi-tree based on the archetype technique that selects the model according to the semantic similarity with respect to the combined model, the number of rules and the test-cost associate to the model. Considering these factors f1, f2 and f3, the exact formula for this combination is:

$$similarity^{f1} \cdot \left(1/_{numrules^{f2}} \right) \cdot \left(1/_{test \cos t^{f3}} \right)$$

These factors affect how the Archetype is extracted. If all the factors except similarity are left to 0, then it is just a semantic extraction, as seen in Section 10.4.

But if these factors are used, then we can extract an archetype taking into account its length and its test cost too.

This permits the user to extract the best single solution according to the user's relevance: comprehensible and shortness (Occam), accuracy and AUC (similarity) and test cost (test cost factor).

When test costs are not known, we can assume a uniform vector, i.e. all the attributes have associated the same cost. This is sometimes similar to the AUC measure of ROC analysis. Assuming a uniform test cost vector we aim at obtaining the model that performs fewer tests.

Finally, test-cost vectors can be used to give more relevance to some attributes than others, e.g. when the user thinks that some attributes can be more useful or comprehensible than others.

As we have said before, ensemble methods require the use of many solutions and this implies the application of nearly all the attributes, thus producing very high test costs. In the following subsection, we will show how to exploit the multi-tree structure and how to get close to the accuracy and AUC levels of combined solutions, but still maintain low test costs and preserve comprehensibility.

11.7.1 Experiments

In this section we present an experimental evaluation of the ideas presented in the previous sections. The results are obtained using the SMILES system.

For the experiments, we used Gain Ratio criterion as splitting criterion and we chose a random method for populating the shared ensemble (after a solution is found, one suspended OR-node is woken at random). By default, we use 100 alternative

OR-nodes for constructing the multi-tree. Pruning was not enabled unless otherwise stated.

Since there are many sources of randomness and since the differences in this kind of comparisons are usually tight, we performed the experiments by averaging 10 results of a 10-fold cross-validation. The results present the mean accuracy of the 10×10 runs for each dataset, and finally the geometric mean of all the datasets.

The first experiment we show in Table 11.3 illustrates the accuracy and test cost obtained for the five datasets whose test cost vectors are known (taken from the UCI repository). Table 11.4 includes the results in AUC. We show the results for the first solution of the multi-tree, for the ensemble (combination), for the Occam solution and for an archetype solution computed with f1=15 (Occam), f2=4 (semantic similarity), and f3=1 (test cost). The results are quite clear. The ensemble solution requires the application of many tests, and consequently it has a quite high test cost. The Occam solution gives no better results than the first solution. However, the archetype solution does improve slightly the initial result in accuracy or AUC, and test cost.

	1st Sol		Ensemble		Archetype		Occam	
#	Acc.	T. cost	Acc.	T. cost	Acc.	T. cost	Acc.	T. cost
11	99.23	37.36	99.31	44.57	99.16	35.77	99.24	35.55
12	52.08	346.76	54.76	455.8	52.74	340	52.12	345.6
17	75.80	14.74	81.80	41.80	75.87	13.87	75.20	14.26
23	64.44	34.62	67.76	38.46	64.68	33.96	64.12	34.09
26	69.63	24.56	72.74	34.85	70.28	23.66	69.01	26.41
GMean	70.63	43.86	73.82	64.75	70.99	42.31	70.33	43.61

Table 11.3: Accuracy and test cost for 5 datasets using the original splitting criterion

In the next tables 11.5 and 11.6 we perform the same experiments but using the modified splitting criterion. In this case, the results are extremely positive. Test cost is reduced about twice with respect to the results of the previous table while the accuracy and AUC are not drastically affected. With this modification, the best results, in terms of test cost, are obtained by the first solution. Archetype obtains similar results in AUC and accuracy, however the test cost of this classifier is slightly worse than the first solution.

Finally, since 5 datasets are too few to obtain a conclusive resolution, we have used uniform vectors for the 28 datasets presented in Appendix A. A uniform vector is a kind of assumption that makes it possible to estimate the behaviour of a model for non-uniform test cost vector, in a similar way as ROC analysis is used for evaluating classifiers when the misclassification cost matrix is not known. In tables 11.7 and 11.8

	1st Sol		Ensemble		Archetype		Occam	
#	AUC	T. cost	AUC	T. cost	AUC	T. cost	AUC	T. cost
11	0.998	37.36	0.999	44.57	0.997	35.77	0.998	35.55
12	0.667	346.7	0.692	455.8	0.675	340	0.666	345.6
17	0.718	14.74	0.813	41.80	0.744	13.87	0.749	14.26
23	0.667	34.62	0.710	38.46	0.667	33.96	0.666	34.09
26	0.761	24.56	0.786	34.85	0.767	23.66	0.757	26.41
GMean	0.753	43.86	0.793	64.75	0.762	42.31	0.759	43.61

Table 11.4: AUC and test cost for 5 datasets using the original splitting criterion

	1st Sol		Ensemble		Archetype		Occam	
#	Acc.	T. cost	Acc.	T. cost	Acc.	T. cost	Acc.	T. cost
11	98.06	19.98	98.41	32.19	98.00	20.82	98.30	22.34
12	50.37	213.8	52.97	357.5	50.17	214.1	50.23	221
17	75.67	5.86	82.47	34.81	77.13	8.29	76.60	8.41
23	64.41	18.26	67.91	33.97	63.68	18.89	63.15	20.61
26	63.96	5.38	67.54	19.37	63.95	5.45	64.76	9.09
GMean	68.78	18.97	72.27	48.33	68.83	20.70	68.85	23.89

Table 11.5: Accuracy and test cost for 5 datasets using the modified splitting criterion

we compare the accuracy and AUC of the first solution, combined model, archetype solution (f1=15, f2=4, f3=1) and Occam hypothesis using the original Gain Ratio splitting criterion. Observing the results, we can appreciate that the best classifier seems to be the archetype if we consider accuracy, AUC and test cost, nonetheless the differences with the Occam model, and the first solution are small.

The same situation occurs when employing the modified version of the splitting criterion (Tables 11.9 accuracy, and 11.10 AUC). Nevertheless, the modified version of the splitting criterion reduces significantly the test cost without degrading too much the quality in AUC or accuracy of the classifiers. As expected the combined solution gets the best results in accuracy and AUC, however the test cost associated to this model is very expensive. The three single models obtain very similar results, although the best trade-off between Accuracy/AUC and test cost is got by the archetype model.

Summing up, we have presented some techniques that can reduce considerably the test cost of applying a classifier without varying significantly the its accuracy. This fact could help to generalise the use of machine learning is contexts where this kind

	1st Sol		Ensemble		Archetype		Occam	
#	Acc.	T. cost	Acc.	T. cost	Acc.	T. cost	Acc.	T. cost
11	0.993	19.98	0.997	32.19	0.993	20.82	0.995	22.34
12	0.656	213.8	0.681	357.5	0.649	214.1	0.654	221
17	0.787	5.86	0.830	34.81	0.762	8.29	0.792	8.41
23	0.671	18.26	0.711	33.97	0.662	18.89	0.665	20.61
26	0.701	5.38	0.729	19.37	0.700	5.45	0.711	9.09
GMean	0.753	18.97	0.782	48.33	0.743	20.70	0.754	23.89

Table 11.6: AUC and test cost for 5 datasets using the modified splitting criterion

of costs is important, for instance to support decisions in medicine.

11.8 Conclusions and Contributions of this Chapter

We have reassessed the construction and evaluation of decision trees based on a very practical and direct way to compute the convex hull of the ROC curve of all the possible labellings of a decision tree. The cost of this operation is just $\mathcal{O}(n \cdot \log n)$, for ordering n leaves of a tree according to their local positive accuracy. This gives a different perspective on decision tree learning, where just clustering trees are learnt, and classes are assigned at application time.

Our approach to using only $n+1$ points is closely related to the ordering of decision tree leaves already presented in [Blockeel and Struyf, 2001] and the ranking of predictions and its use for computing the AUC measure presented in [Hand and Till, 2001]. In comparison with Hand and Till's approach, their AUC measure is equivalent to ours but our node-based way of computing the AUC gives more insight and allows a direct implementation as splitting criterion. This leads to the first splitting criterion based on estimated probabilities we are aware of that is not a weighted average of the impurities of the children. We have extended AUCsplit to more than 2 classes with the generalised M function introduced by [Hand and Till, 2001] Finally, we have also introduced some measures adapted as splitting criterion based on the well-known Mean Squared Error and Log-loss measures.

We have performed an experimental evaluation of the splitting criteria defined in this chapter by comparing the accuracy and AUC evaluations measures of the generated models. The experiments indicate, that in general the results of the MSE and LogLoss gives similar, or even slightly better results than the popular Gain Ration for the AUC measure and accuracy.

We would like to point out that, while we have focused on decision trees in this chapter, the results can be equally used with other learning methods that partition the instance space, such as CN2 or many ILP systems, or our FLIP system.

Regarding test cost, we have studied two ways of reducing the cost of testing attributes which is required by a classifier. First, we have introduced an adaptation of the splitting criterion that takes into account the cost associated to an attribute. In this way, the trees generated by this splitting criterion considerably reduce the test cost without affecting too much the accuracy and AUC. On the hand, we have also investigated a new version of the archetype method that extracts the model according to the semantic similarity with respect to the combined solution, the number of rules, and the test cost. With this, we overcome the problems of ensemble methods: much fewer resources, comprehensibility, and highly reduced the cost of tests.

	1st Sol		Ensemble.		Archetype		Occam	
#	Acc.	T. cost	Acc.	T. cost	Acc.	T. cost	Acc.	T. cost
1	77.94	3.83	82.06	3.97	78.02	3.84	77.53	3.84
2	93.81	4.55	95.01	7.81	93.68	4.23	94.04	4.38
3	92.43	4.95	93.45	14.88	92.63	4.78	92.39	4.71
4	88.61	2.91	91.52	4.97	88.96	3.28	88.77	2.93
5	99.61	4.86	99.34	9.28	99.52	5.26	99.61	4.92
6	48.29	6.09	49.52	7.01	48.39	6.04	47.97	6.16
7	91.47	4.44	94.22	14.27	92.42	4.62	93.14	4.46
8	38.23	6.05	37.08	6.84	37.77	6.02	37.85	6.03
9	66.33	2.87	68.23	2.96	65.87	2.82	65.87	2.86
10	74.06	2.97	77.25	3.82	75.38	2.93	73.81	2.94
11	52.08	7.72	54.76	9.98	52.67	7.72	52.12	7.69
12	75.80	4.38	81.80	13.48	77.27	4.67	75.20	4.67
13	62.72	5.26	67.47	11.14	64.17	5.33	62.78	5.26
14	78.53	4.84	83.33	11.47	79.47	4.92	79.14	4.86
15	94.74	2.94	95.56	7.64	94.67	3.30	94.37	3.13
16	94.13	1.89	94.80	3.53	94.47	2.04	94.27	1.81
17	64.44	4.83	67.76	5.69	64.68	4.74	64.12	4.83
18	95.22	3.17	99.78	5.56	99.93	2.77	99.96	2.72
19	70.97	5.16	76.65	5.99	72.62	5.08	69.82	5.05
20	97.47	2.56	97.71	5.85	97.73	2.55	97.60	2.54
21	91.95	4.01	92.95	4.71	93.19	3.20	93.05	3.43
22	69.63	4.65	72.74	6.17	69.45	4.71	69.01	4.82
23	62.25	5.41	62.13	7.77	64.00	5.34	64.75	5.34
24	95.91	4.08	96.31	7.96	96.08	4.31	96.11	4.23
25	60.60	4.49	62.40	4.85	61.13	4.32	60.60	4.41
26	99.23	2.96	99.31	5.14	99.20	3.03	99.24	2.83
27	77.16	4.74	81.95	6.92	77.80	4.72	78.16	4.71
28	93.00	4.23	93.29	11.06	92.76	3.87	93.41	3.90
GMean	76.73	4.13	78.95	6.89	77.32	4.12	76.94	4.07

Table 11.7: Accuracy and test cost with Gain Ratio splitting criterion

	1st Sol		Ensemble		Archetype		Occam	
#	AUC	T. cost	AUC	T. cost	AUC	T. cost	AUC	T. cost
1	0.824	3.83	0.804	3.97	0.818	3.84	0.821	3.84
2	0.979	4.55	0.985	7.81	0.978	4.23	0.978	4.38
3	0.968	4.95	0.980	14.88	0.966	4.78	0.968	4.71
4	0.953	2.91	0.939	4.97	0.948	3.28	0.950	2.93
5	1.000	4.86	1.000	9.28	0.999	5.26	1.000	4.92
6	0.657	6.09	0.675	7.01	0.659	6.04	0.655	6.16
7	0.991	4.44	0.998	14.27	0.992	4.62	0.994	4.46
8	0.622	6.05	0.643	6.84	0.618	6.02	0.620	6.03
9	0.641	2.87	0.669	2.96	0.635	2.82	0.638	2.86
10	0.900	2.97	0.942	3.82	0.890	2.93	0.886	2.94
11	0.667	7.72	0.692	9.98	0.674	7.72	0.666	7.69
12	0.718	4.38	0.813	13.48	0.747	4.67	0.749	4.67
13	0.724	5.26	0.788	11.14	0.717	5.33	0.713	5.26
14	0.823	4.84	0.862	11.47	0.831	4.92	0.826	4.86
15	0.987	2.94	0.988	7.64	0.985	3.30	0.985	3.13
16	0.987	1.89	0.994	3.53	0.988	2.04	0.988	1.81
17	0.667	4.83	0.710	5.69	0.665	4.74	0.666	4.83
18	0.987	3.17	1.000	5.56	1.000	2.77	1.000	2.72
19	0.710	5.16	0.765	5.99	0.722	5.08	0.674	5.05
20	0.991	2.56	0.991	5.85	0.991	2.55	0.990	2.54
21	0.970	4.01	0.991	4.71	0.978	3.20	0.977	3.43
22	0.761	4.65	0.786	6.17	0.761	4.71	0.757	4.82
23	0.559	5.41	0.372	7.77	0.542	5.34	0.578	5.34
24	0.997	4.08	0.998	7.96	0.997	4.31	0.997	4.23
25	0.763	4.49	0.785	4.85	0.763	4.32	0.769	4.41
26	0.998	2.96	0.999	5.14	0.998	3.03	0.998	2.83
27	0.873	4.74	0.914	6.92	0.878	4.72	0.884	4.71
28	0.976	4.23	0.993	11.06	0.977	3.87	0.976	3.90
GMean	0.833	4.13	0.843	6.89	0.833	4.12	0.833	4.07

Table 11.8: AUC and test cost with Gain Ratio splitting criterion

	1st Sol		Ensemble		Archetype		Occam	
#	Acc.	T. cost	Acc.	T. cost	Acc.	T. cost	Acc.	T. cost
1	78.03	3.78	82.29	3.97	78.47	3.72	77.74	3.79
2	93.77	4.31	95.01	7.80	94.01	3.89	93.26	4.05
3	91.29	3.91	93.48	12.62	92.20	3.78	92.48	3.79
4	88.61	2.91	91.52	4.97	88.96	3.28	88.77	2.93
5	99.61	4.86	99.34	9.28	99.52	5.26	99.61	4.92
6	47.10	5.37	48.87	6.43	47.55	5.39	47.35	5.51
7	91.47	4.44	94.22	14.27	92.42	4.62	93.14	4.46
8	38.23	6.05	37.08	6.84	37.77	6.02	37.85	6.03
9	66.37	2.75	68.70	2.96	66.43	2.70	65.73	2.77
10	74.06	2.97	77.25	3.82	75.38	2.93	73.81	2.94
11	52.08	7.72	54.76	9.98	52.67	7.72	52.12	7.69
12	75.53	4.33	81.00	12.93	75.73	4.43	74.67	4.50
13	62.72	5.26	67.47	11.14	64.17	5.33	62.78	5.26
14	78.53	4.84	83.33	11.47	79.47	4.92	79.14	4.86
15	94.74	2.94	95.56	7.64	94.67	3.30	94.37	3.13
16	95.33	1.67	96.20	3.54	94.87	1.75	95.27	1.74
17	63.41	3.87	68.91	5.51	65.09	3.74	64.56	3.92
18	95.22	3.17	99.78	5.56	99.93	2.77	99.96	2.72
19	70.97	5.16	76.65	5.99	72.62	5.08	69.82	5.05
20	97.47	2.56	97.71	5.85	97.73	2.55	97.60	2.54
21	93.14	2.93	92.48	4.68	94.29	2.56	93.67	2.85
22	69.39	3.73	72.71	5.49	69.28	3.72	69.01	3.93
23	62.25	5.41	62.13	7.77	64.00	5.34	64.75	5.34
24	96.06	3.26	96.63	6.72	96.08	3.34	96.16	3.37
25	62.73	4.12	64.53	4.80	63.00	4.03	61.53	4.14
26	99.11	2.20	99.21	4.21	99.10	2.31	99.19	2.22
27	77.16	4.74	81.95	6.92	77.80	4.72	78.16	4.71
28	93.53	3.31	92.41	10.44	92.12	3.28	93.47	3.30
GMean	76.75	3.82	79.06	6.67	77.37	3.81	76.98	3.82

Table 11.9: Accuracy and test cost using the modified Gain Ratio splitting criterion

	1st Sol		Ensemble		Archetype		Occam	
#	AUC	T. cost	AUC	T. cost	AUC	T. cost	AUC	T. cost
1	0.835	3.78	0.819	3.97	0.835	3.72	0.834	3.79
2	0.979	4.31	0.985	7.80	0.978	3.89	0.977	4.05
3	0.964	3.91	0.976	12.62	0.965	3.78	0.968	3.79
4	0.953	2.91	0.939	4.97	0.948	3.28	0.950	2.93
5	1.000	4.86	1.000	9.28	0.999	5.26	1.000	4.92
6	0.652	5.37	0.675	6.43	0.656	5.39	0.653	5.51
7	0.991	4.44	0.998	14.27	0.992	4.62	0.994	4.46
8	0.622	6.05	0.643	6.84	0.618	6.02	0.620	6.03
9	0.638	2.75	0.666	2.96	0.638	2.70	0.639	2.77
10	0.900	2.97	0.942	3.82	0.890	2.93	0.886	2.94
11	0.667	7.72	0.692	9.98	0.674	7.72	0.666	7.69
12	0.718	4.33	0.804	12.93	0.759	4.43	0.752	4.50
13	0.724	5.26	0.788	11.14	0.717	5.33	0.713	5.26
14	0.823	4.84	0.862	11.47	0.831	4.92	0.826	4.86
15	0.987	2.94	0.988	7.64	0.985	3.30	0.985	3.13
16	0.990	1.67	0.996	3.54	0.988	1.75	0.990	1.74
17	0.672	3.87	0.717	5.51	0.683	3.74	0.683	3.92
18	0.987	3.17	1.000	5.56	1.000	2.77	1.000	2.72
19	0.710	5.16	0.765	5.99	0.722	5.08	0.674	5.05
20	0.991	2.56	0.991	5.85	0.991	2.55	0.990	2.54
21	0.974	2.93	0.995	4.68	0.976	2.56	0.971	2.85
22	0.761	3.73	0.785	5.49	0.754	3.72	0.761	3.93
23	0.559	5.41	0.372	7.77	0.542	5.34	0.578	5.34
24	0.997	3.26	0.998	6.72	0.997	3.34	0.997	3.37
25	0.783	4.12	0.799	4.80	0.776	4.03	0.769	4.14
26	0.996	2.20	0.998	4.21	0.996	2.31	0.997	2.22
27	0.873	4.74	0.914	6.92	0.878	4.72	0.884	4.71
28	0.977	3.31	0.994	10.44	0.971	3.28	0.978	3.30
GMean	0.834	3.82	0.843	6.67	0.835	3.81	0.835	3.82

Table 11.10: AUC and test cost with the modified Gain Ratio splitting criterion

Chapter 12

Conclusions and Future Work

In this final chapter, we summarise the main contributions of this thesis and we point out several directions for further research.

12.1 Conclusions

A system can be considered to have intelligent behaviour if it can adapt its conduct according to the observations taken from a given context. The degree of intelligence is determined by the system's level of adaptation to the context. For instance, most birds avoid feeding from a field when there is a person, but only the most intelligent are able to ignore a scarecrow. Not only individuals can show intelligent behaviour, groups of individuals also need to take decisions according to their operating context.

Nowadays, organisations must operate in a computerised scenario, where the observations, actions and any other kind of information are stored in databases. As a result, companies, governments and even individuals usually deal with huge amounts of data about their context and their inner operation. This information is very useful for predicting future situations, and for making correct decisions in accordance with these predictions. However, the huge amounts of data make manual analysis impossible. In this scenario, computational models generated by data-mining techniques have proven to be very effective in assisting humans to make their decisions. An important factor in the success of data mining has been the incorporation of machine learning methods to extract models from data.

Machine learning techniques automatically extract knowledge from data by expressing the information in a fixed format, or representation language. This representation language is one of the key factor in the success of the learning process. The language must be expressive enough to represent data with a complex structure (normalised relational databases, XML or web documents), or to express complex models (relational associations, recursive patterns, etc.). However, while rich expressiveness permits the handling of problems with an inherent complex structure, the learning algorithm is more complicated and less efficient. The application of machine learning methods that employ a very rich, expressive representation language (relational learning) for problems with a high number of examples and/or attributes becomes very problematic. In these cases, approaches that use more limited representation languages, such as propositional rules, often obtain good results with limited resources. Other methods, such as neural networks, ensemble methods or support vector machines give significantly more accurate models, but they are treated as black-boxes. In any case, the comprehensibility of the learned model is an essential factor for its successful application. Consider the scenario where data-mining tools interact with humans. In some application fields related to decision support systems such as medical decision making, the comprehensibility of the models is essential. More dramatically, "Knowledge Discovery" cannot be understood without comprehensible models.

The common nexus of the contributions presented in this dissertation is the definition of learning techniques that can be easily applied in the context where they are needed. In this regard, we have strived to develop supervised learning methods that produce accurate and comprehensible models from past experiences that minimise the costs of generation and the costs of application. The user of these methods must be able to provide the system with restrictions about the way in which the models should be generated (resources, number of models, comprehensibility, etc.) and to provide information about the way in which models should behave at the moment of application (cost matrices, attribute costs, throughput). In this direction, machine learning techniques could be seamlessly used in data mining applications.

In short, the main contributions of this thesis are:

1. A framework for inductive functional logic programming.

   Modern multi-paradigm declarative languages combine functional and logic programming styles by employing narrowing as operational semantics. Functional logic programming provides some mechanisms, such as types and higher order that logic programming lacks. A classification problem can be expressed naturally by a function whose output is a nominal type. Inspired by inductive logic programming, we have introduced a framework for the automatic learning of functional logic programs. To the best of our knowledge, this is the first framework for the learning of functional logic programs based on narrowing as operational semantics. An incremental version of the framework has also been defined in order to make the algorithm more effective. This learning algorithm has been implemented in the FLIP system. Some experiments have shown the capability of the system to induce problems with a complex structure. In particular, IFLP framework deals more naturally than ILP with multi-class problems, positive-data only problems, or semi-structured problems such as XML documents.

2. A learning method to obtain a set of accurate and comprehensible solutions

   We have defined a learning method that employs a structure called decision multi-tree. The main difference between decision multi-trees and decision trees is that the former can contain a set of hypotheses simultaneously. We have studied some strategies to generate the multi-tree based on a beam search, and then to extract one single model (a decision tree), which usually improves the accuracy and comprehensibility of the model generated by classical decision tree methods. Consequently, the decision multi-tree learning algorithm can be considered an *anytime* algorithm, because the precision and comprehensibility of the generated model is improved in accordance with to the amount of resources consumed.

3. Shared ensemble, an ensemble method that optimises resource consumption.

   Ensemble methods obtain an accurate model by combining the prediction of a set of models. This use of combination can be considered an enrichment of the representation language. Since a multi-tree holds a set of hypotheses, we have also investigated the combination of such hypotheses as an ensemble method. This method inherits the advantages of ensemble methods, i.e. the combined model is often more accurate than any of the hypotheses in the ensemble. However, in a multi-tree, the common parts of the decision trees are shared. Therefore, the consumption of resources is importantly meliorated with respect to other ensemble methods such as *Boosting* or *Bagging*. Consequently, shared ensembles significantly reduce one of the major obstacles for the application of ensemble methods in real problems. Despite the reduction in resources of the shared structure, one serious drawback still remains: the loss of comprehensibility.

4. A new technique for extracting a comprehensible model from an ensemble

   As we have stated, accuracy is significantly increased by using ensemble methods. Nevertheless, these techniques have important limitations, mainly the loss of comprehensibility of the combined model and the large amount of memory required to store the ensemble of hypotheses. The high consumption of computational resources is partially solved by using shared ensembles. However, this learning method still generates an incomprehensible model. In order to overcome this problem and to improve the use of resources even more, we have defined an original method that extracts a single hypothesis (called archetype) from the ensemble so that this hypothesis is similar semantically to the combined model. In this way, the archetype model constitutes a comprehensible model that does not lose too much accuracy with respect to the combined hypothesis. Moreover, the extracted model can be employed as a comprehensible explanation of the combined models. The experiments demonstrate that the archetype model significantly increases the accuracy over single decision trees. At the same time, it does not have the drawbacks of a combined model: poor comprehensibility, slow throughput and high consumption of resources to store the ensemble. In this regard, archetyping can be considered as one of the more accurate approaches for generating a comprehensible model.

Finally, let us remark that the extraction of the archetype model can also be performed with respect to an external oracle. This fact opens an interesting path towards other applications of the technique, for instance, the use of the archetype as a rule extractor from artificial neural networks.

5. Cost-sensitive decision-tree construction

Since in many cases not all misclassifications have the same cost, accuracy is not the best measure for evaluating classifiers. Total cost (derived from the confusion and the cost matrices) is a much better way. Nevertheless, cost information is often unknown until application time. In these cases, an optimal way to evaluate classifiers when cost circumstances are unknown or uncertain is ROC analysis. ROC analysis provides tools to distinguish classifiers that are optimal under certain class and cost distributions from classifiers that are always sub-optimal. We have defined a method that permits the efficient computation of the Area Under the ROC Curve (AUC) of a binary (two-class) classifier formed by disjoint rules. This computation is based on the study of the optimal labellings of such rules according to the cost context. The technique can be applied to all the learning methods that produce disjoint rules, such as IFLP, ILP, decision-tree learning, rule learning, etc.

For decision-tree learning, we have employed this mechanism of computing the AUC to define a splitting criterion that, to the best of our knowledge, is the first splitting criterion based on estimated probabilities that is not a weighted average of the impurities of the children.

With regard to multi-class problems, another notable contribution is the definition of new splitting criteria with the aim of constructing decision trees that improve the AUC measure.

We have also studied some strategies to generate decision trees that minimise the test cost of their attributes. We have considered a generation of the multi-tree according to the test cost by a modification of the splitting criterion. This new version reduces the test cost without overly degrading the accuracy/AUC of the classifier. Finally we are able to obtain an archetype that looks for a compromise between AUC, length, and test cost.

In summary, this dissertation represents one step forward in the hard and long pursuit of making machine learning tools easier to be successfully applied. More expressiveness, more comprehensibility, fewer resources, more precision and lower application costs are all desirable features in this direction. From this intercourse, we have integrated different topics on machine learning, such as cost-sensitive learning, ensemble methods and relational data mining. From a methodological point of view, we have considered some conceptual developments with systematic empirical evidence.

This work has produced several learning methods. These methods have been implemented on two systems: FLIP and SMILES. Let us describe the natural applications where these systems can be employed. FLIP can be used for learning from data with a complex structure, or semi-structured data, such as XML documents. It can also be applied to problems with recursive and relational concepts, and for software engineering applications (program synthesis, debugging...) and generally to small and medium-sized datasets. SMILES can be used for knowledge discovery from databases (data mining) with medium and large datasets. It can also be applied to related applications, such as decision-making support, especially where context is essential and possibly variable: medicine, campaign design, mail filtering, fraud detection, etc.

12.2 Future Work

There are several interesting future lines of research. Let us enumerate some of them:

1. Learning functional logic programs by top-down induction of decision trees

   As a first task, we propose the extension of our system, SMILES, to learn functional logic programs and supersede the FLIP system. The general idea is to determine a set of partitions which is able to introduce features such as calls to background functions, higher-order features or recursive calls. With this goal in mind, we should study new splitting criteria that could guide the search, especially, if complex structure is allowed in the output term (not only a nominal class). The FOIL system [Cameron-Jones and Quinlan, 1994; Quinlan and Cameron-Jones, 1995; Quinlan, 1996c] employs a similar algorithm, (i.e., an algorithm based on decision trees to learn first-order programs), but it has very significant limitations. Apart from the drawbacks which are inherent to ILP, FOIL requires positive evidence with close-world assumption, literals are not permitted to contain function symbols (this reduces the hypothesis search space) and recursion is also limited. Our intention is to use higher-order features and the notions of transformation and feature [Flach and Lavrac, 2000; Bowers *et al.*, 2002]. The use of the multi-tree structure, which can consider multiple alternatives in the search, may be crucial in this pursuit. For more expressive languages, it is very difficult for eager policies to reach the target functions.

2. More effective criterion to populate the multi-tree

   Section 10.2 has introduced the combination of the set of hypotheses in the multi-tree as an ensemble method. We have studied some suspended OR-node selection criteria and a random method has been selected as a trade-off between accuracy and efficiency. In this line of research, we are interested in establishing a suspended OR-node selection criterion for generating decision trees. This probably should be based on the semantic discrepancy of classifiers because combination accuracy depends not only on the number of classifiers but also on the variety. This technique would provide a way to improve the results of the ensemble method with fewer iterations.

   We also propose the study of fusion and transformation techniques for improving the combination of the shared ensemble in terms of AUC. In this dissertation, we have included the AUC results in all the experiments. However, the techniques (except for the MAUC, MSE and Logloss splitting criteria) have been designed to improve the accuracy of classifiers. This is also a must in many other machine learning techniques because not much research on these topics has been performed (due to the novelty of the AUC measure in machine learning).

3. Extracting the archetype analytically

   Chapter 10.5 has presented archetyping as a successful method for extracting an accurate and comprehensible model from an ensemble. This model is semantically similar to the combined hypothesis. An invented random dataset (formed usually by thousands) of examples is generated and then labelled by the combined hypothesis in order to establish the similarity between each model and the combination. This step slows down the archetype technique.

   As future work along this line, we propose the study of a technique for extracting the archetype analytically from the multi-tree. This extraction could be performed in a way similar to the one presented in this thesis, but without the requirement of using a reference dataset.

4. AUC measure for multi-class problems

   The method presented in Chapter 11 is an effective way to compute the AUC value of classifiers formed by disjoint rules for two-class problems. With regard to multi-class problems, the exact calculation is generally intractable in terms of computational cost. Although some estimations to compute the AUC measure for multi-class problems have been proposed ([Hand and Till, 2001; Fawcett, 2001]), the real precision of such approaches has not yet been studied. As a very ambitious work, we propose extending our method of computing the actual AUC to multi-class problems. A study of the quality of the existing approximations for computing AUC in multi-class problems with respect to the exact AUC value could also be very interesting.

Finally, another future task might be the development of a global AUC search heuristic, which would compute the optimality of a split taking into account not only the leaves in the split, but also all the other opened leaves of the tree. We think that a monotonic AUC-based heuristic could be derived in order to implement an optimal AO* search.

Bibliography

Abiteboul S., Buneman P., and Suciu D., 1999. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers.

Aha D.W., Lapointe S., Ling C.X., and Matwin S., 1994. Inverting Implication with Small Training Sets. In F. Bergadano and L. de Raedt, editors, *Proceedings of the European Conference on Machine Learning*, volume 784 of *LNAI*, pages 31–48. Springer, Berlin.

Alpuente M., Ballis D., Correa F., and Falaschi M., 2002. A Multi Paradigm Automatic Correction Scheme. In *Proc. of 11th Int'l Workshop on Functional and (Constraint) Logic Programming*.

Alpuente M., Ballis D., Correa F., and Falaschi M., 2003. Automated Correction of Functional Logic Programs. In *Proc. of Proc. of European Symposium on Programming (ESOP 2003)*. To appear.

Antoy S., 1992. Definitional Trees. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming; Third International Conference; Proceedings*, pages 143–157. Springer-Verlag, Berlin, Germany.

Antoy S., Echahed R., and Hanus M., 2000. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822.

Arts T. and Giesl J., 1996. Termination of Constructor Systems. In *Proc. of 7th Int. Conf. on Rewr. Tech. and Appl. (RTA-96)*, LNCS, pages 63–77.

Arts T. and Giesl J., 2000. Termination of Term Rewriting using Dependency Pairs. *Theoretical Computer Science*, 236(1–2):133–178.

Bauer E. and Kohavi R., 1999. An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants. *Machine Learning*, 36(1/2):105–139.

Bergadano F. and Gunetti D., 1993. An Interactive System to Learn Functional Logic Programs. In *Proc. of the 13th Int. Joint Conf. on Art. Int.*

Bernardo J.M. and Smith A.F.M., 1994. *Bayesian Theory.* Wiley and Sons, Chichester.

Blake C. and Merz C., 1998. UCI Repository of machine learning databases.

Blockeel H., 1999. Top-down Induction of First Order Logical Decision Trees. *AI Communications*, 12:119–20.

Blockeel H. and Raedt L.D., 1998. Top-down Induction of First Order Logical Decision Trees. *Artificial Intelligence*, 101(1-2):285–297.

Blockeel H. and Struyf J., 2001. Frankenstein Classifiers: Some experiments on the Sisyphus data set. In C. Giraud-Carrier, N. Lavrač, and S. Moyle, editors, *ECML/PKDD'01 workshop notes*, pages 1–12.

Booker L.B., Goldberg D.E., and Holland J.H., 1989. Classifier Systems and Genetic Algorithms. *Artificial Intelligence*, 40:235–282.

Bowers A.F., Giraud-Carrier C., Kennedy C., Lloyd J.W., and Mackinney-Romero R., 1997. A framework for Higher-Order Inductive Machine Learning. In *Representation issues in reasoning and learning.* CompulogNet.

Bowers A.F., Giraud-Carrier C., and Lloyd J.W., 2000. Classification of Individuals with Complex Structure. In *Proceedings of 7th Int. Conf. on M. L. (ICML'2000)*, pages 81–88. Morgan Kaufmann.

Bowers A.F., Giraud-Carrier C., and Lloyd J.W., 2002. A Knowledge Representation Framework for Inductive Learning. Http://discus.anu.edu.au/ jwl/.

Bradford J.P., Kunz C., Kohavi R., and Brunk C., 1998. Pruning Decision Trees with Misclassification Costs. *Lecture Notes in Computer Science*, 1398.

Bratko I., Muggleton S., and Karalič A., 1998. Applications of Inductive Logic Programming. In R. Michalski, I. Bratko, and M. Kubat, editors, *Machine Learning and Data Mining.* John Wiley and Sons Ltd., Chichester.

Breiman L., 1996. Bagging predictors. *Machine Learning*, 24(2):123–140.

Breiman L., Friedman J.H., Olshen R.A., and Stone C.J., 1984. *Classification and Regression Trees.* Wadsworth Publishing Company.

Buntine W., 1990. *A Theory of Learning Classification Rules.* Ph.D. thesis, School of Computing Science in the University of Technology, Sydney.

Buntine W., 1993. Learning classification trees. In D.J. Hand, editor, *Artificial Intelligence frontiers in statistics*, pages 182–201. Chapman & Hall,London.

Cameron-Jones R. and Quinlan J.R., 1994. Efficient Top-down Induction of Logic Programs. *SIGART Bulletin*, 5(1):33–42.

Casella G. and Berger R.L., 1990. *Statistical Inference*. Wadsworth and Brooks/Cole, Pacific Grove, CA.

Cendrowska J., 1987. PRISM: An algorithm for inducing modular rules. *International Journal of Man-Machines Studies*, 27:349–370.

Clark P. and Niblett T., 1989. The CN2 Induction Algorithm. *Machine Learning*, 3:261–283.

Cohen J., 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Meas.*, 20:37–46.

Cohen W., 1994. Recovering Software Specifications with ILP. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 142–148. MK.

Conklin D. and Witten I., 1994. Complexity-Based Induction. *Machine Learning*, 16:203–225.

Correa F., 2002. *Depuración Declarativa de Programas Lógico Funcionales*. Ph.D. thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia.

Cristianini N. and Schöllkopf B., 2000. Support Vector Machines and Kernel Methods: The New Generation of Learning Machines. *The AI Magazine*, 23(3):31–41.

De Raedt L., 1998. Attribute Value Learning versus Inductive Logic Programming: The Missing Links (Extended Abstract). In D. Page, editor, *Proc. of the 9th International Workshop on Inductive Logic Programming, ILP'98*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 1–8.

Dean T. and Boddy M., 1988. An analysis of time-dependent planning. In *Proc. of the 7th National Conference on Artificial Intelligence*, pages 49–54.

Dershowitz N., 1987. Termination of Rewriting. In J.P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 69–115. Academic Press. Reprinted from *Journal of Symbolic Computation*.

Dietterich T.G., 2000a. Ensemble Methods in Machine Learning. In *First International Workshop on Multiple Classifier Systems*, pages 1–15.

Dietterich T.G., 2000b. An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization. *Machine Learning*, 40(2):139–157.

Dietterich T.G. and Bakiri G., 1995. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286.

Domingos P., 1999. MetaCost:A General Method for Making Classifiers Cost-Sensitive. In S. Chaudhuri and D. Madigan, editors, *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 155–164. ACM Press, N.Y.

Drummond C. and Holte R., 2000. Explicitly Representing Expected Cost: An Alternative to ROC Representation. In *Knowledge Discovery and Data Mining*, pages 198–207.

Duda R.O. and Hart P.E., 1974. *Pattern Classification and Scene Analysis*. John Wiley and Sons, New York.

Dzeroski S. and Lavrac N., editors, 2001. *Relational Data Mining*. Springer-Verlag.

Echahed R., 1990. On completeness of narrowing strategies. *Theoretical Computer Science*, 72(2-3):133–146.

Egan J., 1975. Signal Detection Theory and ROC analysis. In *Seried in Cognitition and Perception*. Academic Press.

Elkan C., 2001. The Foundations of Cost-Sensitive Learning. In B. Nebel, editor, *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 973–978. Morgan Kaufmann Publishers, Inc., San Francisco, CA.

Esposito F., Malerba D., and Semeraro G., 1997. A Comparative Analysis of Methods for Pruning Decision Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(5):476–491.

Estruch V., Ferri C., Hernández J., and Ramírez M., 2002a. Re-designing Cost-Sensitive Decision Tree Learning. In *Workshop of Data Mining and Learning at the 8th Iberoamerican Conference on Artificial. Intelligence, Iberamia'02*, pages 33–42.

Estruch V., Ferri C., Hernández J., and Ramírez M., 2002b. Shared Ensembles using Multi-trees. In *the 8th Iberoamerican Conference on Artificial. Intelligence, Iberamia'02*, volume 2527 of *Lecture Notes in Computer Science*, pages 204–213.

Estruch V., Ferri C., Hernández J., and Ramírez M., 2002c. SMILES: A Multi-purpose Learning System. In *Logics in Artificial Intelligence, European Conference, JELIA*, volume 2424 of *Lecture Notes in Computer Science*, pages 529–532.

Estruch V., Ferri C., Hernández J., and Ramírez M., 2002d. Un Sistema para la Extracción de Conocimiento en Bioinformática. In *Workshop Bioinformatics and Artificial Intelligence at the 8th Iberoamerican Conference on Artificial. Intelligence, Iberamia'02*, pages 77–87.

Estruch V., Ferri C., Hernández J., and Ramírez M., 2003. Beam Search Extraction and Forgetting Strategies on Shared Ensembles. In *4th International Workshop on Multiple Classifier Systems, MCS2003*, Lecture Notes in Computer Science. To appear.

Evgeniou T. and Pontil M., 2001. Support Vector Machines: Theory and Applications. *Lecture Notes in Computer Science*, 2049.

Fan W., Stolfo S., Zhang J., and Chan P., 1999. AdaCost: misclassification cost-sensitive boosting. In *Proc. 16th International Conf. on Machine Learning*, pages 97–105. Morgan Kaufmann, San Francisco, CA.

Fawcett T., 2001. Using Rule Sets to Maximize ROC Performance. In *2001 IEEE International Conference on Data Mining (ICDM-01)*.

Fayyad U., 2001. Knowledge Discovery in Databases: An Overview. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 28–47. Springer-Verlag.

Ferri C., Flach P., and Hernández J., 2002a. Learning Decision Trees using the Area Under the ROC Curve. In C. Sammut and A. Hoffmann, editors, *Proceedings of the 19th International Conference on Machine Learning*, pages 139–146. Morgan Kaufmann.

Ferri C., Hernández J., and Ramírez M., 2000a. Aprendizaje de Programas Lógico Funcionales. *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial*, 20:34–52.

Ferri C., Hernández J., and Ramírez M., 2000b. The FLIP system Homepage. `http://www.dsic.upv.es/~flip/`.

Ferri C., Hernández J., and Ramírez M., 2000c. Learning functional logic classification concepts from databases. In *9th International Workshop on Functional and Logic Programming (WFLP'2000)*, pages 78–95.

Ferri C., Hernández J., and Ramírez M., 2001a. AND/OR Trees for generating Functional Logic Programs. In *Proc. Joint Conf. on Declarative Programming, AGP2001*, pages 329–342.

Ferri C., Hernández J., and Ramírez M., 2001b. Incremental Learning of Functional Logic Programs. In *Proc. of the Fifth Int. Symposium on Functional and Logic Programming*, volume 2024 of *Lecture Notes in Computer Science*, pages 231–247.

Ferri C., Hernández J., and Ramírez M., 2001c. Learning MDL-guided Decision Trees for Constructor-Based Languages. In *Work In Progress Track, Conference of Inductive Logic Programming, ILP2001*, pages 39–50.

Ferri C., Hernández J., and Ramírez M., 2002b. From Ensemble Methods to Comprehensible Models. In *The 5th International Conference on Discovery Science*, volume 2534 of *Lecture Notes in Computer Science*, pages 164–177.

Ferri C., Hernández J., and Ramírez M., 2002c. Induction of Decision Multi-Trees using Levin Search. In *Int. Conf. on Computational Science, ICCS'02*, volume 2329 of *Lecture Notes in Computer Science*, pages 166–175.

Ferri C., Hernández J., and Ramírez M., 2002d. SMILES System, a multi-purpose learning system. `http://www.dsic.upv.es/~flip/smiles/`.

Ferri C., Hernández J., and Ramírez M., 2003. A Framework for Learning Functional Logic Programs. *New Generation Computing*. Accepted subject to revision.

Flach P., 2000. The use of functional and logic languages in machine learning. In M. Alpuente, editor, *Ninth International Workshop on Functional and Logic Programming (WFLP2000)*, pages 225–237. Universidad Politecnica de Valencia.

Flach P., Blockeel H., Ferri C., Hernández J., and Struyf J., 2003. Decision Support for Data Mining:Introduction to ROC analysis and its applications. In M. Bohanec, editor, *Data Mining and Decision Support: Integration and Collaboration*. Kluwer Academic Publishers, Boston.

Flach P. and Lavrac N., 2000. The role of feature construction in inductive rule learning. In L.D. Raedt and S. Kramer, editors, *Proceedings of the ICML2000 workshop on Attribute-Value and Relational Learning: crossing the boundaries*, pages 1–11. 17th International Conference on Machine Learning, Stanford, USA.

Flener P. and Yilmaz S., 1999. Inductive Synthesis of Recursive Logic Programs: Achievements and Prospects. *Journal of Logic Programming*, 41:141–195.

Fostel G., 1995. Evolutionary Computation. Toward a New Philosophy of Machine Intelligence. *IEEE Press*.

Freund Y. and Schapire R.E., 1996. Experiments with a new Boosting algorithm. In *Proc. 13th International Conference on Machine Learning*, pages 148–146. Morgan Kaufmann.

Freund Y. and Schapire R.E., 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139.

Fribourg L., 1985. Slog: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. 2nd IEEE Int'l Symp. on Logic Prog.*, pages 172–185. IEEE.

Gini C., 1955. Variabilitá e mutabilita. *Memorie di metodologia statistica*.

Hand D., 1997. *Construction and Assessment of Classification Rules*. John Willey & son.

Hand D. and Till R., 2001. A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems. *Machine Learning*, 45(2):171–186.

Hans W., Loogen R., and Winkler S., 1992. On the Interaction of Lazy Evaluation and Backtracking. In M. Bruynooghe and M. Wirsing, editors, *Proc. Int'l Symp. on Programming Languages: Implementations, Logics and Programs, PLILP'92*, pages 355–369. Springer-Verlag.

Hansen L.K. and Salamon P., 1990. Neural Network Ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-12(10):993–1001.

Hanus M., 1994. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19-20:583–628.

Hanus (ed.) M., 2000. Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`.

Hernández J. and Ramírez M., 1998. Inverse Narrowing for the Induction of Functional Logic Programs. In *Proc. Joint Conf. on Declarative Programming, AGP'98*, pages 379–393.

Hernández J. and Ramírez M., 1999. A Strong Complete Schema for Inductive Functional Logic Programming. In *Proc. of the 9th Int. Workshop on Inductive Logic Programming, ILP'99*, volume 1634 of *LNAI*, pages 116–127.

Hernández J. and Ramírez M., 2001. Predictive Software. *Automated Software Engineering*, 8, n.2.

Ho T., 1998. C4.5 Decision Forests. In *Proc. of 14th Intl. Conf. on Pattern Recognition, Brisbane, Australia*, pages 545–549.

Hoche S. and Wrobel S., 2001. Relational Learning Using Constrained Confidence-Rated Boosting. In C. Rouveirol and M. Sebag, editors, *Proceedings of the 11th International Conference on Inductive Logic Programming*, volume 2157 of *Lecture Notes in Artificial Intelligence*, pages 51–64. Springer-Verlag.

Hullot J., 1980. Canonical Forms and Unification. In *5th Int'l Conf. on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag.

Hussmann H., 1986. Unification in Conditional-Equational Theories. Technical report, Fakultät für Mathematik und Informatik, Universität Passau.

Ichise R., 1998. Synthesizing Inductive Logic Programming and Genetic Programming. In *Proc.the 13th Europ. Conf. on A. I. (ECAI-98)*, pages 467–468.

Jaffar J. and Lassez J.L., 1987. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM.

Japkowicz N., 2000. The Class Imbalance Problem: Significance and Strategies. In *Proceedings of the 2000 International Conference on Artificial Intelligence (IC-AI'2000)*, volume 1, pages 111–117.

J.D. Boissonat M.Y., 1998. *Algorithmic Geometry.* Cambridge University Press.

Jensen R. and Schlimmer J., 1996. An Anytime Solution to Inductively Learn Decision Trees. In *AAAI Fall Symposium on Flexible Computation in Intelligent Systems*, pages 106–110.

Jorge A. and Brazdil P., 1995. Architecture for Iterative Learning of Recursive Definitions. In *Proc. of 5th Int. Workshop on Ind. Logic Progr.*, pages 95–108.

Julian P. and Moreno G., 1996. Nuevas Semanticas para Programas Logicos Funcionales. Technical report, Department of Information Systems and Computation, Valencia University of Technology.

Kearns M. and Mansour Y., 1996. On the boosting ability of top-down decision tree learning algorithms. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 459–468.

Kietz J.U. and Wrobel S., 1992. Controlling the Complexity of Learning in Logic through Syntactic and Task-oriented Models. In *Inductive Logic Programming.* Academic Press.

King R., Muggleton S., Lewis R., and Sternberg M., 1992. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences of the USA*, 89(23):11322–11326.

Klop J., 1992. Term Rewriting Systems. *Handbook of Logic in Computer Science*, I:1–112.

Knoll U., Nakhaeizadeh G., and Tasend B., 1994. Cost-Sensitive Pruning of Decision Trees. In F. Bergadano and L. de Raedt, editors, *Proceedings of the European Conference on Machine Learning*, volume 784 of *LNAI*, pages 383–386. Springer, Berlin.

Kohavi R. and Kunz C., 1997. Option decision trees with majority votes. In *Proc. 14th International Conference on Machine Learning*, pages 161–169. Morgan Kaufmann.

Kukar M. and Kononenko I., 1998. Cost-Sensitive Learning with Neural Networks. In H. Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 445–449. Chichester.

Kuncheva L.I., 2002. A Theoretical Study on Six Classifier Fusion Strategies. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 24(2):281–286.

Kuncheva L.I. and Whitaker C.J., 2002. Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. Submitted to Machine Learning.

Lane T., 2000. Extensions of ROC analysis to Multi-Class Domains. In *ICML-2000 Workshop on cost-sensitive learning.*

Lau C., editor, 1992. *Neural Networks: Theoretical Foundations and Analysis.* IEEE.

Lavrac N. and Dzeroski S., 1994. *Inductive Logic Programming: Techniques and Applications.* Artificial Intelligence. Ellis Horwood (Simon & Schuster).

Lavrac N., Dzeroski S., and Grobelnik M., 1991. Learning Nonrecursive Definitions of Relations with LINUS. In Y. Kodratoff, editor, *Proceedings of the European Working Session on Learning : Machine Learning (EWSL-91)*, volume 482 of *LNAI*, pages 265–281. Springer Verlag.

Lavrac N., Gamberger D., and Turney P.D., 1996. Cost-sensitive feature reduction applied to a hybrid genetic algorithm. *Lecture Notes in Computer Science*, 1160:127–134.

Li M. and Vitányi P., 1997. *An Introduction to Kolmogorov Complexity and its Applications.* 2nd Ed. Springer-Verlag.

Liu B., Xia Y., and Yu P.S., 2000. Clustering Through Decision Tree Construction. In *CIKM*, pages 20–29.

Lloyd J., 1995. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol.

Lux W., 2001. Adding Linear Constraints over Real Numbers to Curry. *Lecture Notes in Computer Science*, 2024:185–195.

Maclin R. and Opitz D., 1997. An Empirical Evaluation of Bagging and Boosting. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 546–551. AAAI Press.

Margineantu D., 1999. Building ensembles of classifiers for loss minimization. In *Proceedings of Interface-99.* The Interface Foundation of North America.

Margineantu D., 2002. *Methods for Cost-Sensitive Learning.* Ph.D. thesis, Oregon State University.

Margineantu D. and Dietterich T., 1997. Pruning adaptive boosting. In *14th Int. Conf. on Machine Learning*, pages 211–218. Morgan Kaufmann.

Margineantu D. and Dietterich T., 2000. Bootstrap Methods for the Cost-Sensitive Evaluation of Classifiers. In *Proc. 17th International Conf. on Machine Learning*, pages 583–590. Morgan Kaufmann, San Francisco, CA.

Mehta M., Rissanen J., and Agrawal R., 1995. MDL-Based Decision Tree Pruning. In *Proc. of the 1st Int. Conf. on Knowledge Discovery and Data Mining (KDD'95)*, pages 216–221.

Michalski R.S., 1969. On the quasi-minimal solution of the general covering problem. In *Proc. Fifth Int. Symposium on Information Processing, FCIP 69*, volume A3. Bled, Yugoslavia.

Michalski R.S., 1983. A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–161.

Michalski R.S., Mozetic I., Hong J., and Lavrac N., 1986. The multi-purpose incremental learning system AQ15 and its testing application on three medical domains. In *Proc. Fifth National Conference on Artificial Intelligence*, pages 1041–1045. Morgan Kaufmann, San Mateo, CA.

Michie D., Muggleton S., Page D., and Srinivasan A., 1994. To the international computing community: A new East-West challenge. Technical report, Oxford University Computing laboratory, Oxford,UK.

Middeldorp A. and Okui S., 1998. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757.

Mingers J., 1988. An Empirical Comparison of Selection Measures for Decision-Tree Induction. *Machine Learning*, 3:319.

Mingers J., 1989. An Empirical Comparison of Pruning Methods for Decision Tree Induction. *Machine Learning*, 4:227.

Mitchell T.M., 1997. *Machine Learning*. McGraw-Hill.

Mizoguchi F., Ohwada H., Daidoji M., and Shirato S., 1996. Learning rules that classify ocular fundus images for glaucoma diagnosis. In S. Muggleton, editor, *Conference of Inductive Logic Programming, ILP1996*, volume 1314 of *Lecture Notes in Artificial Intelligence*, pages 146–162.

Mooney R.J. and Califf M.E., 1995. Induction of First–Order Decision Lists: Results on Learning the Past Tense of English Verbs. *J. of A.I. Research*, 3:1–24.

Mooney R.J. and Richards B., 1992. Automated debugging of logic programs via theory revision. In S. Muggleton, editor, *Proceedings of the 2nd International Workshop on Inductive Logic Programming*, Report ICOT TM-1182.

Morik K., 1997. Knowledge Discovery in Databases — An Inductive Logic Programming Approach. *Lecture Notes in Computer Science*, 1337.

Mossman D. and Somoza E., 1991. ROC curves, test accuracy, and the description of diagnostic tests. *J. Neuropsychiatr. Clin. Neuroscience, 3*, pages 330–3.

Muggleton S., 1991. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318.

Muggleton S., 1995. Inverse Entailment and Progol. *New Generation Computing Journal*, 13:245–286.

Muggleton S., 1999. Inductive Logic Programming: Issues, Results, and the Challenge of Learning Language in Logic. *Artificial Intelligence*, 114(1–2):283–296.

Muggleton S., 2000. Progol v.5.0. `http://www.doc.ic.ac.uk/~shm/Software/`.

Muggleton S. and Buntine W., 1992. Machine Invention of First-Order Predicates by Inverting Resolution. In S. Muggleton, editor, *Inductive Logic Programming*, pages 261–280. Academic Press.

Muggleton S. and Feng C., 1992. Efficient Induction in Logic Programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press.

Muggleton S., King R., and Sternberg M., 1992. Protein secondary structure prediction using logic. In S. Muggleton, editor, *Conference of Inductive Logic Programming, ILP1992*, Report ICOT TM-1182, pages 228–259.

Muggleton S. and Page D., 1994. Beyond first-order learning: Inductive learning with higher-order logic. Technical Report PRG-TR-13-94, Oxford University, Oxford.

Neri F. and Saitta L., 1994. Knowledge Representation in Machine Learning. *Lecture Notes in Computer Science*, 784:20–??

Nilsson N., 1998. *Artificial Intelligence: a new synthesis*. Morgan Kaufmann.

Núñez M., 1988. Economic induction: A case study. In *Proceedings of the Third European Working Session on Learning, EWSL-88*, volume 55, pages 139–145. Morgan Kaufmann.

Núñez M., 1991. The Use of Background Knowledge in Decision Tree Induction. *Machine Learning*, 6:231–250.

Olsson R., 1995. Inductive Functional Programming using Incremental Program Transformation. *Artificial Intelligence*, 74(1):55–81.

Parmanto B., Munro P.W., and Doyle H.R., 1996. Improving Committee Diagnosis with Resampling Techniques. In D.S. Touretzky, M.C. Mozer, and M.E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 882–888. The MIT Press.

Pazzani M., Merz C., Murphy P., Ali K., Hume T., and Brunk C., 1994. Reducing misclassification costs. In *Proc. 11th International Conference on Machine Learning*, pages 217–225. Morgan Kaufmann.

Pearl J., 1985. *Heuristics: Intelligence search strategies for computer problem solving*. Addison Wesley.

Plotkin G., 1971a. *Automatic Methods of Inductive Inference*. Ph.D. thesis, Edinburgh University.

Plotkin G., 1971b. A further Note on Inductive Generalization. *Machine Intelligence*, 6.

Press W.H., Teukolsky S.A., Vetterling W.T., and Flannery B.P., 1992. *Numerical Recipes in C, 2nd. edition*. Cambridge University Press.

Provost F. and Domingos P., 2002. Tree Induction for Probability-Based Ranking. *Machine Learning*, To appear.

Provost F. and Fawcett T., 1997. Analysis and Visualization of Classifier Performance: Comparison under Imprecise Class and Cost Distributions. In D.H. et al., editor, *Proc. of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining (KDD-97)*, page 43. AAAI Press.

Provost F. and Fawcett T., 1998. Robust Classification Systems for Imprecise Environments. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 706–713. AAAI Press, Menlo Park.

Provost F. and Fawcett T., 2001. Robust Classification for Imprecise Environments. *Machine Learning, 42*, pages 203–231.

Quinlan J.R., 1987. Simplifying Decision Trees. *International Journal of Man-Machine Studies*, 27(3):221–234.

Quinlan J.R., 1990a. Induction of Decision Trees. In *Readings in Machine Learning*. Morgan Kaufmann.

Quinlan J.R., 1990b. Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266.

Quinlan J.R., 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA.

Quinlan J.R., 1996a. Bagging, Boosting, and C4.5. In *Proc. of the 30th Nat. Conf. on A.I. and the 8th Innovative Applications of A.I. Conf.*, pages 725–730. AAAI Press / MIT Press.

Quinlan J.R., 1996b. Boosting first-order learning. In S. Arikawa and A.K. Sharma, editors, *Proceedings of the 7th International Workshop on Algorithmic Learning Theory*, volume 1160 of *LNAI*, pages 143–155. Springer, Berlin.

Quinlan J.R., 1996c. Foil v.6. `http://www.cse.unsw.edu.au/~quinlan/foil6.sh`.

Quinlan J.R., 1996d. Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, 5:139–1612.

Quinlan J.R., 1998a. Data Mining Tools See5 and C5.0. http://www.rulequest.com/see5-info.html.

Quinlan J.R., 1998b. MiniBoosting Decision Trees. available at http://www.cse.unsw.EDU.AU/ quinlan/miniboost.ps.

Quinlan J.R., 2001. Relational Learning and Boosting. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 292–306. Springer-Verlag.

Quinlan J.R. and Cameron-Jones R., 1995. Induction of logic programs: FOIL and related systems. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):287–312.

Quinlan J.R. and Rivest R.L., 1989. Inferring Decision Trees Using The Minimum Description Length Principle. *Information and Computation*, 80:227=–248.

R. Kimball L. Reeves M.R. and ThornthWaite W., 1998. *The data warehouse life cycle: expert methods for designing, developing, and deploying data warehouses.* John Willey and Sons.

Raedt L.D., 1992. *Interactive Theory Revision: An Inductive Logic Programming Approach.* Academic Press.

Rao M.K., 1997. A framework for Incremental Learning of Logic Programs. *Theoretical Computer Science*, 185:191–213.

Reddy U., 1985a. Narrowing as the operational semantics of functional languages. In *Proceedings of the IEEE Symposium on Logic Programming, Boston*, pages 138–151.

Reddy U., 1985b. Narrowing as the Operational Semantics of Functional Languages. In *Proc. 2nd IEEE Int'l Symp. on Logic Prog.*, pages 138–151. IEEE.

Richards B.L. and Mooney R.J., 1991. First Order Theory Revision. In *Proc. of the 8th Int. Workshop on Machine Learning*, pages 447–451.

Robinson J.A., 1965. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41.

Rouveirol C., 1994. Flattening and Saturation: Two Representation Changes for Generalization. *Machine Learning*, 14:219–232.

Rumelhart D.E., Hinton G.E., and Williams R.J., 1986. Learning internal representation by error propagation. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. Cambridge, MA: MIT Press.

Shapiro E., 1983. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Mass.

Siekmann J., 1984. Universal Unification. In *7th Int'l Conf. on Automated Deduction*, volume 170 of *Lecture Notes in Computer Science*, pages 1–42.

Smith D., 1984. The synthesis of LISP programs from examples: A survey. *Automatic Program Construction Techniques*, pages 307–324.

Smolka G., 1995. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin.

SPSS, 2002. Clementine Homepage. `http://www.spss.com/clementine/`.

Srinivasan A., 1999. Note on the location of optimal classifiers in n-dimensional ROC space. Technical Report PRG-TR-2-99, Oxford University Computing Laboratory.

Srinivasan A., 2001. Context-Sensitive Models in Inductive Logic Programming. *Machine Learning*, 43(3):301–324.

Srinivasan A., King R., and Bristol D.W., 1999. An Assessment of ILP-Assisted Models for Toxicology and the PTE-3 Experiment. In S. Džeroski and P. Flach, editors, *Proceedings of the 9th International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 291–302. Springer-Verlag.

Srinivasan A., Muggleton S., King R., and Sternberg M., 1994. Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel, editor, *Conference of Inductive Logic Programming, ILP1994*, volume 237 of *GMD-Studien*, pages 217–232. Gesellschaft für Mathematik und Datenverarbeitung MBH.

Swets J., 1986. Measuring the Accuracy of Diagnostic System. *Science*, 240:1285–1293.

Swets J., Dawes R., and Monahan J., 2000. Better Decisions through Science. *Scientific American*, 283(4):82–87.

Tan M., 1993. Cost-Sensitive Learning of Classification Knowledge and Its Applications in Robotics. *Machine Learning*, 13:7–33.

Tan M. and Schlimmer J., 1990. Two Case Studies in Cost-Sensitive Concept Acquisition. In W. Dietterich, Tom; Swartout, editor, *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 854–860. MIT Press.

Togashi A. and Noguchi S., 1990. Inductive Inference of Term Rewriting Systems Realizing Algebras. In *Proc. of the 1st Int. Workshop on Algorithmic Learning Theory*, pages 411–424.

Turcotte M., Muggleton S., and Sternberg M., 1998. Application of Inductive Logic Programming to Discover Rules Governing the Three-Dimensional Topology of Protein Structure. In D. Page, editor, *Conference of Inductive Logic Programming, ILP1998*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 53–64.

Turney P.D., 1995a. Cost-Sensitive Classification: Empirical Evaluation of a Hybrid Genetic Decision Tree Induction Algorithm. *Journal of Artificial Intelligence Research*, 2:369–409.

Turney P.D., 1995b. Low Size-Complexity Inductive Logic Programming: The East-West Challenge Considered as a Problem in Cost-Sensitive Classification. In

L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 247–263. Department of Computer Science, Katholieke Universiteit Leuven.

Turney P.D., 2001. Types of Cost in Inductive Concept Learning. In *Workshop on Cost-Sensitive Learning at the Seventeenth International Conference on Machine Learning*, pages 15–21.

Varsek A., 1993. *Genetic Inductive Logic Programming*. Ph.D. thesis, University of Ljubljana, Slovenia.

Webb G.I., 1996. Further experimental evidence against the utility of Occam's razor. *Journal of Artificial Intelligence Research*, 4:397–417.

Weiss G. and Provost F., 2001. The Effect of Class Distribution on Classifier Learning: an Empirical Study. Technical report, Department of Coputer Science, Rutgers University.

Weiss S. and Indurkhya N., 1998. *Predictive Data Mining: A Practical Guide*. Morgan Kaufmann, San Francisco.

Wikipedia, 2003. Wikipedia, the free encyclopedia. `http://www.wikipedia.org`.

Wilson B., 1998. The Machine Learning Dictionary for COMP9414.

Wrobel S., 1993. On the proper definition of Minimality in Specialization and Theory Revision. In P. Brazdil, editor, *Proc. of ECML-93*, volume 667 of *Lecture Notes in Computer Science*, pages 65–82.

Wrobel S., 1996. First Order Theory Refinement. In L.D. Raedt, editor, *Advances in Inductive Logic Programming*, pages 14–33. IOS Press.

Wrobel S., 2001. Inductive Logic Programming for Knowledge Discovery in Databases. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 74–101. Springer-Verlag.

Appendix A

Datasets

For the experimental evaluation of the methods and techniques that we have presented during the dissertation we have employed 28 datasets from the UCI dataset repository [Blake and Merz, 1998]. This repository contains around a hundred datasets, and it is very popular in the machine learning community. The datasets that conform the collection are widely employed to evaluate the quality of machine learing methods.

Table A shows the name, the size in number of examples, the number of classes and the number of nominal and numerical attributes of the 28 datasets. A short description of each dataset can be found in Table A and Table A.

#	Datasets	Size	Classes	Nom. Attr.	Num. Attr.
1	Balance Scale	325	3	0	4
2	Breast Cancer	699	2	0	9
3	Breast Cancer Wisconsin	569	2	1	30
4	Car Evaluation	1728	4	5	0
5	Chess	3196	2	36	0
6	Contraceptive Method Choice	1473	3	7	2
7	Dermatology	366	6	33	1
8	Echocardio	132	3	1	6
9	Haberman's Survival	306	2	0	3
10	Hayes-Roth	106	3	5	0
11	Heart Disease	920	5	8	5
12	Hepatitis	155	2	14	5
13	Horse-colic-outcome	366	3	14	8
14	Horse-colic-surgical	366	2	14	8
15	House Congressional Voting	435	2	16	0
16	Iris Plan	158	3	0	4
17	Liver-disorders-mod	345	2	0	6
18	MONK's1	566	2	6	0
19	MONK's2	601	2	6	0
20	MONK's3	554	2	6	0
21	New Thyroid	215	3	0	5
22	Pima Indians Diabetes	768	2	0	8
23	Postoperative Patient	90	3	7	1
24	Segmentation Image Database	2310	7	0	14
25	Teaching Assistant Evaluation	151	3	2	3
26	Thyroid ANN	7200	3	15	0
27	Tic-Tac-Toe Endgame	958	2	8	0
28	Wine Recognition	178	3	0	13

Table A.1: Datasets used in the experiments.

#	Datasets	Description
1	Balance Scale	This data set was generated to model psychological experimental results.
2	Breast Cancer	A breast cancer domain was obtained from the University Medical Centre Institute of Oncology, Ljubljana. Thanks go to M. Zwitter and M. Soklic for providing the data.
3	Breast Cancer Wisconsin	A breast cancer databases was obtained from the University of Wisconsin Hospitals, Madison.
4	Car Evaluation	Car Evaluation Database was derived from a simple hierarchical decision model originally developed for the demonstration of an expert system.
5	Chess	King+Rook versus King+Pawn on a7, The pawn on a7 means it is one square away from queening.
6	Contraceptive Method Choice	This dataset is a subset of the 1987 National Indonesia Contraceptive Prevalence Survey. The problem is to predict the current contraceptive method choice.
7	Dermatology	Dataset formed by clinical features of patients. The aim is to determine the type of Eryhemato-Squamous Disease.
8	Echocardio	Dataset composed by patients suffered heart attacks at some point in the past. Te aim is to predict from the other variables whether or not the patient will survive at least one year.
9	Haberman's Survival	The dataset contains cases from a study at the University of Chicago's Billings Hospital on the survival of patients who had undergone surgery for breast cancer.
10	Hayes-Roth	Dataset about Human subjects classification and recognition performance.
11	Heart Disease	Joint dataset that contains 4 databases concerning heart disease diagnosis.
12	Hepatitis	A dataset concerning hepatitis diagnosis.
13	Horse-colic-outcome	The dataset contains cases from a study of survival of horses suffering from colics.
14	Horse-colic-surgical	The dataset contains cases from a study of surgical lesions of horses suffering from colics.

Table A.2: Short description of the datasets used in the experiments (1).

#	Datasets	Description
15	House Congressional Voting	This data set includes votes for each of the U.S. House of Representatives Congressmen on 16 key votes. The aim is to predict the party affiliation of the Congressmen.
16	Iris Plan	The dataset contains 150 instances, where each class refers to a type of iris plant.
17	Liver-disorders	The variables are all blood tests which are thought to be sensitive to liver disorders that might arise from excessive alcohol consumption (class).
18	MONK's1	A set of three artificial domains over the same attribute space. The three MONK problems were the basis of a first international comparison of learning algorithms.
19	MONK's2	
20	MONK's3	
21	New Thyroid	Five lab. tests are used to try to predict whether a patient's thyroid belongs to the class euthyroidism, hypothyroidism or hyperthyroidism
22	Pima Indians Diabetes	This primary tumor domain was obtained from the University Medical Centre Institute of Oncology, Ljubljana. Thanks go to M. Zwitter and M. Soklic for providing the data
23	Postoperative Patient	The classification task of this database is to determine where patients in a postoperative recovery area should be sent to next area.
24	Segmentation Image Database	The instances were drawn randomly from a database of 7 outdoor images. The images were handsegmented to create a classification for every pixel.
25	Teaching Assistant Evaluation	The data consist of evaluations of teaching performance over three regular semesters and two summer semesters of 151 teaching assistant.
26	Thyroid ANN	The problem is to determine whether a patient is hypothyroid. Three classes are built: normal, hyperfunction and subnormal functioning.
27	Tic-Tac-Toe Endgame	This database encodes the complete set of possible board configurations at the end of tic-tac-toe games.
28	Wine Recognition	These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars.

Table A.3: Short description of the datasets used in the experiments (2).

Appendix B

The FLIP System

B.1 Introduction

This appendix describes the FLIP system, an implementation of the Inductive Functional Logic Programming (IFLP) framework presented in Chapter 7. IFLP can be defined as the functional (or equational) extension of ILP.

The system induces functional logic programs (IFLP) from examples. For simplicity, the (positive and negative) examples are expressed as pairs of ground terms or ground equations where the right term is in normal form. Positive examples represent terms that will have to be proved equal using the induced program, whereas negative examples consist of terms that do not have to be proved equal. The approach is based on the idea of the inverse resolution of ILP. Starting from the generalisation of positive examples to include variables as arguments of functions, we defined an inverse narrowing mechanism which selects pairs of equations to obtain new equations. The selection criterion is based on the number of positive examples that can be derived from the equation. Then, if the generated equation covers more positive examples, then it is added to the previously selected equations (probably replacing some of them). This process is repeated until a program (or set of equations) is valid according to some selection criteria. More details about the algorithms can be found in Chapter 7, and Chapter 8.

B.2 The System

The FLIP system is a project built in C, with more than 5000 lines of code that implements the IFLP framework. The system includes a simple parser, a narrowing solver, an inverse narrowing method, and a CRG generator.

The system uses the design and implementation of functional logic programming defined by Alpuente, Julian and Moreno in [Julian and Moreno, 1996].

The FLIP system works with two sets of facts: the positive examples and the negative ones, respectively and optionally background knowledge. Starting from these examples the system applies the IFLP algorithm until a solution program is discovered or it has overpassed a limit number of steps of the main loop of the induction process.

B.2.1 Program Options

The files with the examples and the background theory are passed to the FLIP system by means of arguments and options when it is called. If you execute FLIP with the option −?, it prints out the usage information:

```
[cesar@curret]$ flip -?

######################## FLIP System (v0.7) ########################

usage: ./flip [-t] [-b background_file] [-u] [-e] [-n] [-f] [-p
depth]
[-s steps] positive_examples_file [negative_examples_file]
    -t          Switches off the equation termination filter.
    -b file     Uses the specified file as the background knowledge.
    -u          Uses the union operator in the first step.
    -n          Inverse narrowing with full program clauses
                combination.
    -f          Stops immediately when a program is found that covers
                all the examples.
     -r         Computes the number of positive examples that
                covers the program passed with option -b
    -p depth    Depth limit of the narrowing solver(default 19).
    -s steps    Maximum number of steps of the main induction
                loop(default 10).
    -o file     Writes the induced program in the specified file.
    -i examples Incremental induction.
```

B.2.2 Starting FLIP

Here, we comment the purpose of each option:

- **-t** (*Termination filter*): FLIP provides a simple termination filter that permits to reject some non-terminating equations. Nevertheless, there are cases where it is interesting to generate programs with non-terminating equations, for instance *minus* or *max* functions, see [Ferri *et al.*, 2000b] for details.

- **-b** (*Background Knowledge*): The user specifies to FLIP a file with a background theory that will be used in the induction process.
- **-u** (*Union operator*): FLIP can use, if the user wants, the union operator in the first step of the process. As we argued in Chapter 7, the use of the union operator is not suitable in some cases, because it generates long programs with patches. Consequently, this option should be restricted.
- **-n** (*Inverse narrowing*): The inverse narrowing operator is applied by default between the two best equations of the combined programs. With this option the user commits FLIP to apply inverse narrowing between all the clauses of the pair of selected programs (full program clauses combination).
- **-p** (*Narrowing depth*): The system uses a narrowing solver that performs a depth search. Thus, to eliminate possible infinite loops in non-terminating programs a depth limit is fixed, by default it is 19.
- **-f** (*Stop immediately*): We have found problems with some non-terminating equations, because the filter we have implemented is quite naive (the termination problem is far from being simple). When this option is enabled, FLIP immediately exits whenever it finds a solution program. In this way we avoid the possibility of finding a non-terminating equation before the step is completed.
- **-r** (*Program Coverage*): The system just computes the number of positive examples that a functional logic program covers. The program is specified with the "-o" option.
- **-s** (*Main loop limit*): This option sets a maximum number of steps of the main loop of the induction process, by default the limit is 10.
- **-o** (*Write induced program*): In this case the learned program is written into a file specified by the user.
- **-i** (*Incrementality*): FLIP runs the incremental version of the algorithm. In this case a program is induced with a subset of the examples, and it is revised or evaluated with the rest of examples.

B.3 Examples of using FLIP

In this section we show how FLIP solves some simple problems. We begin illustrating the example of the sum of two naturals:

The target program is:

$$\begin{aligned} sum(X,0) &= X \\ sum(s(X),Y) &= s(sum(X,Y)) \end{aligned}$$

where $0, s(0), s(s(0))$... denote the natural numbers. Consider we have the following file *sum.pex* with positives examples of this problem:

$$\begin{aligned}
sum(0,0) &= 0, \\
sum(s(0),s(0)) &= s(s(0)), \\
sum(0,s(0)) &= s(0), \\
sum(s(s(0)),0) &= s(s(0)), \\
sum(s(0),0) &= s(0), \\
sum(s(0),s(s(0))) &= s(s(s(0))), \\
sum(s(s(0)),s(s(0))) &= s(s(s(s(0)))), \\
sum(s(0),0) &= s(0), \\
sum(s(s(s(0))),s(0)) &= s(s(s(s(0)))), \\
sum(s(s(s(0))),s(s(0))) &= s(s(s(s(s(0)))))
\end{aligned}$$

And the following file *sum.nex* with negative ones:

$$\begin{aligned}
sum(s(0),0) &= 0, \\
sum(0,0) &= s(0), \\
sum(s(0),s(0)) &= s(0), \\
sum(s(0),0) &= s(s(0)), \\
sum(0,s(0)) &= s(s(0))
\end{aligned}$$

Note that the syntax of the files is very simple, just equations separated by commas, without any other type of information: types, mode... So, the next step is just executing FLIP:

```
[cesar@curret]$ flip sum.pex sum.nex
```

The program prints into the standard output some information about the inducing process. First, it shows a line that notifies the user about the generation of CRG equations. When this phase is finished, every equation generated is shown together with its coverage:

```
######################## FLIP System (v0.7) ########################
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
```

```
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
Initial CRG Equations
0 sum(0,0) = 0
1 sum(0,X0) = X0
2 sum(X0,0) = X0
.
.
55 sum(s(s(s(X0))),s(s(X0))) = s(s(s(s(s(X0)))))
56 sum(s(s(X0)),s(X0)) = s(s(s(s(0)))))
57 sum(s(s(X0)),s(X0)) = s(s(s(s(X0))))

coverage of program 0: 1 positives,
coverage of program 1: 2 positives,
coverage of program 2: 4 positives,
```

When all the programs have been shown, FLIP chooses the best two, according to the IFLP criterion, and performs inverse narrowing between them:

```
1 Best program: number 2, 4 positives,
2 Best program: number 7, 3 positives,
 %%%%%%% Inverse Narrowing %%%%%%%%

 Inverse Narrowing between:
sum(X0,0) = X0
sum(X0,s(0)) = s(X0)

 Resulting equation:
sum(X0,s(0)) = sum(s(X0),0)

 Analysed equation:
sum(X0,s(0)) = sum(s(X0),0)
```

The new equations provided by the inverse narrowing process are generalised by the CRG operator:

```
@@@@@@@@@@@ CRG process @@@@@@@@@@@
 CRG Equations generated:
```

```
0 sum(X0,s(0)) = sum(s(X0),0)
1 sum(X0,s(X1)) = sum(s(X0),0)
2 sum(X0,s(X1)) = sum(s(X0),X1)
3 sum(X0,s(0)) = s(sum(X0,0))
4 sum(X0,s(X1)) = s(sum(X0,0))
5 sum(X0,s(X1)) = s(sum(X0,X1))
```

Every new equation forms a program, whose coverage is shown. At the end of the step it is checked whether the best program satisfies the requirements of the target program, if it covers them, the process has ended and FLIP prints the program. First the original one is shown, and then, a filtered version without redundant equations:

```
Best program:
0 sum(X0,s(X1)) = sum(s(X0),X1)
1 sum(X0,0) = X0

 &&&&&&&&&&&&&&&& Solution program &&&&&&&&&&&&&
0 sum(X0,s(X1)) = sum(s(X0),X1)
1 sum(X0,0) = X0
 Number of examples covered: 10, optimality 13.459432

 &&&&&&&&&&&&&&&& Filtered solution program &&&&&&&&&&&&&
0 sum(X0,s(X1)) = sum(s(X0),X1)
1 sum(X0,0) = X0
 Number of examples covered: 10, optimality 13.459432

====> Time used: (0.88      seg.)
```

B.3.1 Incrementality

We can employ the incremental version of the algorithm , with the option "-i":

```
[cesar@curret]$ flip -i 5 sum.pex sum.nex
```

With this option, we indicate FLIP to employ only 5 examples to learn a hypothesis. The rest of examples are used to revise or evaluate the hypothesis, according to the incremental version of the algorithm (see Section 7.5).

The program generated in this example is the same, however, since in this case only 5 examples are used to induce the program and the rest are (in this case) covered, the induction is faster.

```
 &&&&&&&&&&&&&&&& Filtered solution program &&&&&&&&&&&&&
```

```
0 sum(X0,s(X1)) = sum(s(X0),X1)
1 sum(X0,0) = X0
 Number of examples covered: 10, optimality 13.459432

====> Time used: (0.28      seg.)
```

B.3.2 Using Background Knowledge

Consider we want to learn a program for the multiplication of natural numbers. An example of such program is:

$$\begin{array}{lcl} prod(s(X0), X1) & = & sum(prod(X0, X1), X1) \\ prod(0, X0) & = & 0 \end{array}$$

where *sum* is the previous learned function, for the addition of natural numbers. We have the following positive evidence:

$$\begin{array}{lcl} prod(s(0), 0) & = & 0, \\ prod(0, 0) & = & 0, \\ prod(s(0), s(s(s(0)))) & = & s(s(s(0))), \\ prod(s(0), s(0)) & = & s(0), \\ prod(s(s(0)), s(s(0))) & = & s(s(s(s(0)))) \end{array}$$

and the negative evidence is:

$$\begin{array}{lcl} prod(0, 0) & = & s(0), \\ prod(0, s(0)) & = & s(0), \\ prod(s(0), 0) & = & s(s(0)), \\ prod(s(0), s(0)) & = & s(s(0)), \\ prod(s(0), s(0)) & = & 0, \\ prod(s(s(0)), s(0)) & = & s(0), \\ prod(s(s(0)), s(s(0))) & = & s(s(s(0))), \\ prod(s(s(0)), s(s(s(0)))) & = & s(s(s(s(s(s(s(0))))))), \\ prod(s(s(s(0))), s(s(0))) & = & s(s(s(s(s(0))))) \end{array}$$

To induce such a function, it is very advisable to employ the definition of *sum* as background knowledge. The use of a program as background knowledge in the induction algorithm can be specified with the option "-b" and the name of a file with a program with the background knowledge equations; i.e.:

```
./flip -b sum.ple prod.pex prod.nex
```

The FLIP system launches the learning algorithm employing in this case background knowledge, and generates the following output:

```
 &&&&&&&&&&&&&&&& Filtered solution program &&&&&&&&&&&&&
0 prod(s(X0),X1) = sum(prod(X0,X1),X1)
1 prod(0,X0) = 0
2 sum(0,X) = X
3 sum(s(Y),X) = s(sum(Y,X))
 Number of examples covered: 5, optimality 9.523562

===> Time used: (0.58      seg.)
```

which is the goal function. We refer to http://www.dsic.upv.es/ flip/flip/ to obtain more information and examples of the system.

Appendix C

The SMILES System

C.1 Introduction

SMILES (Stunning Multi-purpose Integrated LEarning System) is a machine learning system that integrates many different features from other machine learning techniques and paradigms and presents several innovations in almost all of these features. In particular, it extends classical decision tree learners in many ways (new splitting criteria, non-greedy search, new partitions, extraction of several and different solutions), it has an anytime handling of resources, and has a sophisticated and quite effective handling of costs. In this way, SMILES combines and improves the recent interest in hypotheses combination and cost-sensitive learning (a priori and a posteriori class assignments, ROC analysis) outperforming previous systems in many situations.

This appendix tries to present the system in its present form, serving as a short technical description from the machine learning point of view and also as a user's guide. The intention is that this appendix should be self-contained, so it may happen that some explanation would be redundant with other parts of this dissertation.

The appendix is then organised as follows. In Section C.2 the structure of the system is described in a practical and succinct way. SMILES user's manual is presented in Section C.3, describing separately the general inputs/outputs of the system, the general options, the multi-tree features, the different ways to extract several solutions, the cost-sensitive and ROC features, the archetype procedure and other facilities.

C.2 The Structure of the System

In this section we briefly present the main algorithm and underlying methods of SMILES.

Decision tree learning is a very popular kind of machine learning technique. In a decision tree, each node contains a test on an attribute, each branch from a node represents a possible outcome of the test, and each leaf contains a class prediction. A decision tree is usually induced by recursively replacing leaves by test nodes, starting at the root. Classic decision-tree learners such as CART [Breiman *et al.*, 1984],ID3 [Quinlan, 1990a] or C4.5 [Quinlan, 1993] have given very good results and are currently used in many applications; however, they do not have flexibility with respect to trading result quality for computational resources.

The main algorithm of SMILES is also based on a greedy search in the decision tree space, such as classic decision-tree learners. However, SMILES is able to obtain more than one solution, looking for the best one or combining them in order to improve the overall accuracy or minimise the classification cost. To do this, once a node has been selected to be split (an AND node) the other possible splits at this point (OR nodes) are suspended until a new solution is required. In this way, the search space is an AND/OR tree which is traversed producing an increasing number of solutions for increasing provided time. Since each new solution is built following the construction of a complete tree, our method differs from other approaches such as the *Boosting* method, which induces a new decision tree for each solution. The result is a multi-tree rather than a forest; with the advantage that a multi-tree shares the common parts and the forest does not. We perform a greedy search for each solution, but once the first solution is found the following ones can be obtained taking into consideration a limited computation time. Therefore, our algorithm can be considered *anytime* in a certain way.

Apart from the multi-tree (AND-OR) structure, our system extends the representation language by extending the possible partitions. The types handled by SMILES are:

Type	**Use**
Nominal	For value sets, Booleans and any non-numerical attribute.
Numerical	For integers, real numbers and any numerical attribute. Ordered nominal attributes such as {low, medium, high} are not directly handled and should be substituted by integers.

Table C.1: Types handled by the system.

At the present moment, our system allows the partitions of Table C.2, two of them using negation:

Note that the search of the decision tree space requires the use of several criteria. If the search can be re-activated to explore further solutions then more criteria are

Partition	
Partition X=a, X=b, X=c...	Present in ID3, C4.5
Partition X=a, X$\neq$a	
Partition X=Y, X$\neq$Y	
Partition X<c, X$\geq$c	Present in ID3, C4.5

Table C.2: Partitions allowed in the system.

needed. The main criteria used by SMILES are:

- **Splitting Criterion**: among all the possible partitions (split) which one is selected to open the tree.
- **Suspended OR-node Selection Criterion**: specifies which node to explore for second-best solutions.
- **Suspended Nodes Forgetting**: Should all the suspended nodes retained or some of them can be forgotten?
- **Best Tree Selection Criterion**: specifies which of all of these solutions must be shown if the user wants just the best solution.
- **k-Best Different Solutions Selection criterion**: specifies how to select from the AND/OR tree a set of solutions. If k is equal to 1, then it just selects one solution according to the BestTree Selection Criterion.

C.3 User's Manual

The SMILES source-code and documentation about the system can be downloaded from `http://www.dsic.upv.es/~flip/smiles/`. The package includes the C++ sources and some sample datasets. Once the software downloaded and decompressed, follow the *readme* file and run the shell-script for installation on Unix-like machines. There is also a version ready to be compiled in Windows.

If the installation is successful, you can directly type:

./smiles -?

and you will have the following usage information:

****** SMILES v.2.3.1 (Release Date: 23-August-2002) ******
USAGE:
./smiles file.train [file.test] [file.cost] [file.testcost]

That means that the software has been correctly installed.

C.3.1 SMILES Usage Basics

The current SMILES version is a command-line batch application with little interaction during the learning stage. Almost all interaction is performed through the inputs (mainly the training set, the data set and the options) and the outputs (mainly solution trees and statistics).

The previous usage information suggests that the system must be supplied with some files, at least a training set.

We can run the system by using some of the examples that are included with the distribution, e.g., the playtennis example. If we type:

./smiles samples/playt.train samples/playt.test

Using the simplest options (no combination) in the option file (we will discuss on this) the result may be something similar to this:

```
**** SMILES v.2.3.1 (Release Date: 23-August-2002) ****
ftrain: samples/playt.train
ftest: samples/playt.test
fcost:
ftestcost:
Training Set: "samples/playt.train"
No. of Attributes: 4
Cardinality: 14 examples
Class: 0 ("yes"). Distribution: 9
Class: 1 ("no"). Distribution: 5
Valid options.
Creating the multitree.
Learning begins...
The test set: "samples/playt.test" will be used to evaluate the results
Predicting and preparing statistics
Filling new test probabilities of the leaves of the multi-tree with the Test set
15 examples done.
Showing below the properties of the best single tree
SOLUTION 0: 5 rules
Statistics over test set of length: 15:
Relative Accuracy: 1
AUC Hand: 1
END OF RESULTS
Test dataset destroyed successfully
The multitree has been destroyed successfully
====> Time used (for learning): 0.01 secs.
**** Smooth end of SMILES execution ****
```

If we take a look at the results we observe several parts. A first part informs us that the files have been read and tells us about their characteristics: number of attributes of the dataset, number of the examples and class distribution, in the previous example 4 attributes and 14 examples. Finally, it includes statistics on the tree

Let us use a different example: "house-votes":

./smiles samples/house-votes.train samples/house-votes.test

If we use an option file that uses combination, the output varies slightly:

```
**** SMILES v.2.3.1 (Release Date: 23-August-2002) ****
ftrain: samples/house-votes.train
ftest: samples/house-votes.test
fcost:
ftestcost:
Training Set: "samples/house-votes.train"
No. of Attributes: 16
Cardinality: 217 examples
Class: 0 ("democrat"). Distribution: 136
Class: 1 ("repub"). Distribution: 81
Valid options.
Creating the multitree.
Learning begins...
Last Opened Node #1000 learned of 1000
MeanDepth of Second Solution Start Positions: 4.36837
The test set: "samples/house-votes.test" will be used to evaluate the results
Predicting and preparing statistics
Showing first the results of the combination method of all branches:
COMBINATION RESULTS:
Statistics over test set of length: 218:
Relative Accuracy: 0.963303
AUC (example by example) in Hand & Till's way: 0.990568
Filling new test probabilities of the leaves of the multi-tree with the Test set
218 examples done.
FIRST SOLUTION: (solution as if 1 tree were generated).
Note: if postpruning is enabled this solution is not the same as with 1 tree.
Num. of Rules: 21
Statistics over test set of length: 218:
Relative Accuracy: 0.949541
AUC Hand: 0.980477
Showing below the properties of the best single tree
SOLUTION 0: 21 rules
```

Statistics over test set of length: 218:
Relative Accuracy: 0.940367
AUC Hand: 0.976968
END OF RESULTS
Test dataset destroyed successfully
The multitree has been destroyed successfully
====> Time used (for learning): 10.49 secs.
****** Smooth end of SMILES execution ******

Now, learning begins exploring not only the first splits but many others (# of opened OR-nodes, in the previous example 1000), in order to obtain a good result. After the learning process, some statistics of the combination all the trees are shown (0.963 accuracy and 0.991 AUC), statistics of the first solution as if the multi-tree would have only 1 solution (21 rules, 0.950 accuracy and 0.980 AUC) and, finally, a solution which is extracted using a criterion from the multi-tree (21 rules, 0.940 accuracy and 0.977 AUC).

C.3.2 Simple Input and Output Format

Let us explain first the format of the training set file. The lines that begin with ! are meant to be directives to the parser. Only the "!TYPES:" directive is mandatory. After several lines of directives the dataset starts with the values of each argument separated by commas. Each example must be put in a different line. The last value of each line is the class of the example and must not be followed by comma. The meaning of the directives is given in Table C.3:

Directive	**Syntax**	**Mandat.**
!TYPES:	Natural numbers: for nominal types, each number ≥ 1 denotes a different type. For numerical types, a 0 must be , used (0u or OU if missing values are to be taken into account).	Yes
!NAMES:	Argument names separated by commas. Only used for visualising the solutions.	No
!WEIGHTS:	Set of pairs "classname = weight" where weight is a real number. Tries to give more relevance to some classes over others. This will be further explained along with the cost-sensitive options.	No
!RECLASSIFY	Set of pairs "originalclassname>changedclassname;". The semicolon is mandatory even for the last assignment. This options is to change the name of one or more class. This is used for joining two or more classes.	No

Table C.3: Meaning of the directives of the options file.

Let us illustrate this syntax with an example (playt.train):

!TYPES:1,2,3,4,5

```
!NAMES:sky,temp,humid,wind,play
!WEIGHTS:yes=1,no=2
overcast,hot,high,weak,yes
rain,mild,high,weak,yes
rain,cool,normal,weak,yes
sunny,mild,normal,strong,yes
overcast,mild,high,strong,yes
overcast,hot,normal,weak,yes
sunny,hot,high,weak,no
sunny,hot,high,strong,no
rain,cool,normal,strong,no
sunny,mild,high,weak,no
overcast,cool,normal,strong,yes
sunny,cool,normal,weak,yes
rain,mild,normal,weak,yes
rain,mild,high,strong,no
```

This is a dataset with two classes, where all the attributes are nominal and different (this is why all the integers of the directive !TYPES are different). This training set gives more relevance to class "no" (2) than to class "yes" (1) by using the directive "!WEIGHTS".

Another example could be like this.

```
!TYPES:1,2,2,0,3,0,4
!NAMES:sex,fatherstudlevel,motherstudlevel,IQ,region,fameconlevel,studlevel
!RECLASSIFY:elementary>no-uni;secondary>no-uni;highschool>no-uni;
female,secondary,highschool,120,Valencia,20000,doctor
male,university,university,130,Madrid,50000,highschool
female,secondary,elementary,150,Berlin,30000,doctor
male,elementary,elementary,90,Bretagne,40000,secondary
female,secondary,elementary,105,Calabria,15000,university
...
```

In this case we have four nominal and two numerical attributes. Two attributes ("fatherstudlevel" and "motherstudlevel") are of the same type. The class originally had five possible values from the datasets, but the !RECLASSIFY directive has joined the first three into a new class "no-uni" and the remainder two into a new class "uni". This means that this problem will be treated by SMILES as a two classes problem. Note that when the !RECLASSIFY option is used, the type for the class should not be used for any other attribute (using type 2 for the class in this case could produce some errors).

We can ignore some attributes, either nominal or numerical, (if we think they are not useful for learning) when instead of the type, an 'I' or 'i' is put in the !TYPES directive.

Nominal missing values are represented by "?" and are simply considered as an additional value for an attribute. Consequently, tree partitions may have "?" in some of the branches.

For numerical missing values (which must also be represented by "?"), there are three possible options for handling them: ignore any example that contains numerical missing values (by using the value "0" in the !TYPES directive), treat them as a special value (by using the value "0U" or "0u" in the !TYPES directive) or substitute missing numeric values by a 0. We will see how to change this option later.

For instance::

!TYPES:1,2,2,0,I,0U,4
!NAMES:sex,fatherstudlevel,motherstudlevel,IQ,region,fameconlevel,studlevel
female,secondary,highschool,120,Valencia,20000,doctor
male,university,university,?,Madrid,50000,highschool
female,secondary,elementary,150,Berlin,?,doctor
male,elementary,elementary,?,Bretagne,?,secondary
female,?,elementary,105,Calabria,15000,university
...

Now we have some examples with missing numerical and nominal values. We also see that the "region" attribute is ignored and consequently not used for learning. Supposing we have set in the options file that missing values should be discarded and not substituted by zero (we will go back on this), the first example does not contain missing values; the second example is discarded as a missing value appears for an attribute (IQ) that does not accept it; the third example is taken into account because the missing value appears for an attribute (famconlevel) that accepts it; the fourth is discarded as a missing value appears for an attribute (IQ) that does not accept it; the fifth is taken into account and the second attribute (fatherstudlevel) would have an additional value "?".

Numerical partitions with missing values are sets of the form $(X < a, X >= a, X = ?)$[1].

The test set has the same format as the training set, although directives are ignored. If the number or type of the attributes in the test set is different from the training set, an error is produced and SMILES exits. Moreover, it should be noted that in the current version if the test set presents for a nominal type an attribute value that was not present in the training set, an error is produced and SMILES exits.

[1]If two nominal arguments are set to be of the same type, it must be ensured that both of them have missing values or none of them. Otherwise, the types would not be identical and there may be an error.

A very easy way to avoid this is to add a line like the following one in the training set:

?,?,?,?,?

which adds an *unknown* attribute value ? for all types. Then the test dataset should be modified in order to substitute any new value by a ?.

C.3.3 Options File

Before, we saw that the system options are not given through the command line. In order to specify the system options, a configuration system is utilised, called "**options.cfg**".

In this file, comment lines begin with '%'. The rest are pairs "option=value", where spaces should not be placed around the symbol "=". Most of the options are not mandatory but the file "options.cfg" must be placed in the same directory SMILES is.

The syntax of the options file is illustrated by the following excerpt:

```
%–Expected error method: ways to compute expected error
expected error method=no compute
%
%–frequency error smoothing:use smoothing or not
smoothing method=no smoothing
```

As we have said before, lines that begin with '%' are considered comments and are ignored. The options file is then constituted by assignments, where the left hand side is the option name and the right hand side is the value given to the option. Both the option name and the option value side have a strict syntax and no spaces can be inserted on either side of the '=' sign.

Let us make a change in the options file. If we open it with an editor and modify the "show all k-best solutions" option as follows:

```
%–show all k-best solutions
show all k-best solutions=show
```

And now, following with the playtennis example, we run SMILES again with the same training set and test set, we have that now the output includes one solution in form of rules:

```
**** SMILES v.2.3.1 (Release Date: 23-August-2002) ****
...
Let us show the solution: 0
f(X0, X1, X2, X3) = R :- X0=overcast. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=high. [class: no]
```

f(X0, X1, X2, X3) = R :- X0=sunny, X2=normal. [class: yes]
...
****** Smooth end of SMILES execution ******
This shows the solution in the form of conditional rules of the type *Head :- Body*.

C.3.4 General Options

Given the basics of the system, now we are going to describe some other main features and options.

One of the first things to choose in a decision tree learning algorithm is the splitting criterion. Different splitting criteria have been implemented: Left first, Gain (entropy), Gain_Ratio and C4.5, CART , MDL, DKM, Expected_Error and Area Under the ROC Curve.

In the options file, the user can choose among all of these:

%–splitting criterion: criterion which is used to select the best split
%splitting criterion=left first
%splitting criterion=gain
%splitting criterion=gain ratio
%splitting criterion=c4.5
%splitting criterion=cart
%splitting criterion=mgini
%splitting criterion=desc mdl
%splitting criterion=dkm
%splitting criterion=split expected error
%splitting criterion=local roc area
%splitting criterion=one point local roc area
splitting criterion=mse
%splitting criterion=logloss
%splitting criterion=sqdiff
%splitting criterion=genentropy
%splitting criterion=rocv
%splitting criterion=auch
%splitting criterion=aucs

Their description is as follows:

- **left first**: just chooses the first split.
- **gain**: Quinlan's information gain.
- **gain ratio**: Quinlan's information gain ratio.
- **c4.5**: same as gain ratio but splits with gain lower than the mean are discarded.

- **cart**: simple implementation of GINI criterion of CART system.
- **mgini**: true implementation of GINI criterion.
- **desc mdl**: a criterion based on MDL.
- **dkm**: An optimisation of mgini.
- **split expected error**: The split with lowest expected error.
- **local roc area**: the split with greatest area under the ROC curve. Only valid for 2 classes
- **one point local roc area**: simplification of "local ROC area". Just computes the area with one point.
- **MSE**: minimum squared error.
- **logloss**: logloss measure.
- **sqdiff**: computed as the square of the difference between probability for class a and probability for class b. Only valid for two classes.
- **genentropy**: gain and gini can be seen as special cases of a generalised entropy function depending on a power. This is a parametrised split criterion where this exponent can be modified (by program).
- **rocv**: another (not very successful) extension to the roc split for more than 2 classes.
- **auch**: Based on Hand and Till's extension of Area Under the ROC Curve for more than two classes.
- **aucs:** Fawcett's variant of AUCH.

The best results are usually obtained with the MSE, AUCH (MAUC) and c4.5 criteria. CART only works well for two classes, and MGINI should be used instead. MGINI works well for two classes, so never use CART.

Another important feature to be considered in a decision tree learning algorithm is pruning. Currently, SMILES includes pre-pruning methods and post-pruning. The available pre-pruning criteria can be chosen in the options file as follows:

%–pre-pruning method: criterion for pruning when constructing the tree
pre-pruning method=no pruning
%pre-pruning method=proportional
%pre-pruning method=expected error pruning
%pre-pruning method=MDL pruning

%pre-pruning method=stump pruning

%pre-pruning method=pep pruning

%

%–only for stump pruning. Depth Limit.

%–If stump pruning is not active, this option is ignored

stump pruning limit=3

It is advisable to use some kind of pruning when some noise is expected in the data. Many of these criteria have some parameters although currently only the stump pruning (depth limit) is modifiable through the options file, whereas the rest of them can be modified in the source code. The effectiveness on accuracy of these criteria is very variable. The MDL pruning criterion can only be used if the MDL splitting criterion has also been selected.

Post-pruning is less efficient (because the entire tree is populated before pruning) but it is easier to use and more effective. Currently, the only post-pruning method implemented is the "Pessimistic Error Pruning" (PEP) introduced by [Quinlan, 1987]. According to the study in [Esposito *et al.*, 1997], this is the best method that does not modify the tree structure (unlike C4.5 pruning). Although it has a tendency to underprune, we think that it is a quite simple and effective method. The way to activate it is:

%–post-pruning method: criterion for pruning after constructing the tree (not used for combination)

%post-pruning method=no pruning

post-pruning method=pep pruning

It is possible to combine pre-pruning and post-pruning. It could be good results to have a very strict pre-pruning criterion (only pruning very clear cases) and then post-pruning. We have not, though, performed any experiment about this.

Additionally, numerical splits can also be particularised. The problem with numerical splits arises when a dataset has a great number of different values for an attribute. This happens very often with continuous values, such as real numbers. A dataset of thousands of examples and a single numerical attribute could generate a partition with thousands of children. Given n different values, after ordering them, we would have a middle value a_i and a condition $(x < a_i)$ for each interval generated by two consecutive different values. This could slow down the system dramatically. For this reason there are some methods for reducing the number of intervals (also called discretisation methods): in a logarithmic way, with a maximum or with no limit. These options can be selected in the options file:

%–numeric interval criterion: how numerical attributes are handled

numeric interval criterion=log

%numeric interval criterion=max

%numeric interval criterion=no limit

Some of these options have parameters. Currently, these parameters cannot be specified through the options file and must be changed in the source code through program re-compilation.

Finally, the options that correspond to the handling of missing numeric values are:

%– how to handle missing numeric values

missing numeric values=ignore examples with missing numeric values

%missing numeric values=substitute missing numeric values

The first one ignores any example that contains a value '?' for a numerical attribute. The second option substitutes them by 0s. Both options are not applicable if we use the types "0U" or "0u" in the "!TYPES" directive of the training set.

Until now, we have described the options that are common to many decision tree algorithms. In the next sections we describe more distinct and advanced features of SMILES.

C.3.5 Multi-tree Options

As we have discussed in the introduction, one way to overcome the greedy behaviour of traditional decision tree learning is to explore other splits. Once these alternative trees (OR trees) are grown, then the best tree can be selected from all the open branches or a combined solution of all of them could be built.

The first thing to specify is the number of trees in the multi-tree that are going to be explored. More precisely, the number corresponds to how many different OR-nodes will be explored, which can give many more solutions (all their possible combinations). This is done in the options file as follows:

%–multitree: construction: how many or-nodes are opened in the multi-tree

multitree number of trees=1000

Additionally to the number of trees, the way in which second solutions are found has to be also specified. For instance, the following lines would select that the second solution would be the still unexplored node that is *topmost*.

%–second tree opening:how to select the 2nd node to explore

%second tree opening=split optimality

%second tree opening=optimality rival ratio

%second tree opening=optimality rival ratio depth

%second tree opening=optimality rival ratio component

%second tree opening=optimality rival ratio component random

second tree opening=second topmost

%second tree opening=second bottommost

%second tree opening=second random

%second tree opening=second random depth

It is advisable to use topmost to improve accuracy, although it is the one that requires more resources. In case of resource limitations, a good option might be random.

The number of trees (or-nodes) to be explored and how to select them determines the resulting multi-tree. Now, with all these open branches two things can be done: select a subset of solutions or combine them.

If we decide to select one solution, then we have to choose a selection criterion to specify which of all the possible solutions must be shown. To do this, the options file includes the following alternatives:

%–multitree: best tree selection criterion

multitree best tree criterion=occam best

%multitree best tree criterion=test cost best

%multitree best tree criterion=occam and test cost best

%multitree best tree criterion=coverage best

%multitree best tree criterion=cross coverage

%multitree best tree criterion=expected error best

%multitree best tree criterion=split optimality best

%multitree best tree criterion=mdl best

For instance, "occam best" selects the shortest solution possible from the multi-tree. Others such "coverage best", "expected error best" and "mdl best" choose the tree that gets best results with these measures. These options are explained in Section C.3.14. Later on we will see another method for obtaining a single solution from the set (ensemble) of solutions.

Taking into account alternative solutions provides better solutions than the first eager, greedy, single solution. This behaviour may require a lot of memory to store all the alternatives (used or not). In fact, most of the alternate trees (the OR nodes) that are suspended never are woken and, consequently, are never explored. A way to do a more efficient use of resources, can be based on not opening all the OR nodes or, seen in other way, to forget part of the suspended nodes. Currently, SMILES presents "suspended nodes forgetting" methods:

%–suspended nodes forgetting: must all suspended nodes maintained?

suspended nodes forgetting=maintain all

%suspended nodes forgetting=maintain const random

%suspended nodes forgetting=maintain log random

%suspended nodes forgetting=maintain log random with depth

%suspended nodes forgetting=maintain log random with squared depth

%suspended nodes forgetting=maintain log random with depth adjusted

The meaning of these options are:

- **maintain all**: as before. Everything is kept in memory.

- **maintain const random**: a constant number of children (randomly selected) is maintained.
- **maintain log random**: only a log (number of children) are maintained.
- **maintain log random with depth**: only a log(number of $children/(depth+1)$) are maintained.
- **maintain log random with depth adjusted**: as "maintain log random with depth" but a constant is added to the number of children. This constant is hardwired in the program.
- **maintain log random with squared depth**: similar to the preceding ones, but the formula is now: log(number of children/sqrt(depth+1)).

With these methods, memory and time requirements for the multi-tree can be significantly improved. A compromise between memory and quality of results is obtained through the method "maintain log random with depth adjusted". As we have said, the method "maintain all" just maintains all the suspended nodes and it is the recommended option when there are not memory restrictions.

Finally, although most of the methods maintain a logarithm proportion, in some cases the number of remaining nodes must be too low. In order to avoid this, a minimum number of nodes that must be preserved can be specified. This can be done through the following option.

%–suspended nodes maintain const value
suspended nodes maintain const value=2

According to the experimental results shown in Section 10.3.1, *log* is the most economical option, both in memory and in time, and results are not significantly deteriorated.

C.3.6 Combination and Fusion Options

The other way to use the multiplicity in the multi-tree is to combine the results of different branches. There are also several methods to combine the different solutions that can be given in the multi-tree, and they can be selected using the option file.

%– Combination: How to combine several solutions
%multitree solution combination=no combination
%multitree solution combination=cross coverage combination
%multitree solution combination=majority crisp
multitree solution combination=majority absolute stochastic
%multitree solution combination=majority relative stochastic
%multitree solution combination=majority cost stochastic

As we will discuss later, the "majority absolute stochastic" method gives the better results and can be combined with other more sophisticated options.

In particular, a decision tree learner can be seen as a soft classifier, taking the proportion of examples that have fallen in each leaf as the probability that a new examples would be assigned to that class.

When we have multiple solutions the question is how these vectors can be "fusioned" in order to give a mixture or ensemble classifier. In our case the fusion is made whenever two or more OR-nodes are found and their predictions are weighted.

At each leaf node, we have a cardinality vector that tells how many examples of the training set have fallen for each class. The first thing that we can decide is to work with the absolute vector or with the relative vector. For this we have the following options:

%– Combination vector: absolute (n. of examples) or relative (frequency)
combination vector=absolute
%combination vector=relative

In either case, in some situations it may be convenient to perform a Laplace smoothing of these vectors. This can be enabled through the following option:

%– Combination smoothing. Use smoothing before combination
%combination smoothing=true
combination smoothing=false

The vectors can be left as they are originally or they can be modified in several ways, so affecting the resulting fusion.

%– Combination vector method: how to derive the vector
combination vector method=original
%combination vector method=good loser
%combination vector method=bad loser
%combination vector method=difference
%combination vector method=majority
%combination vector method=squared

The exact definition of these transformations is included in Section 10.2.

Finally, with these vectors and each time two or more OR-nodes exist at the same level the following fusion methods can be chosen:

%– Combination fusion method: how to combine the vectors
%combination fusion method=sum
%combination fusion method=prod
%combination fusion method=arithmean
%combination fusion method=geomean
combination fusion method=max
%combination fusion method=min

%combination fusion method=median

The meaning of these fusion strategies can be explained in Section 10.2.

Another thing to take into account is that because of the very nature of combination (combination avoids overfitting and hence improves accuracy and other quality measures), pruning is not beneficial. Consequently, pruning is not recommended to be used for combination.

For this reason, since sometimes we may want post-pruning for the best single solution but not for the combination, there exists another option that disables post-pruning for combination despite it can be enabled for single solutions. This is obtained through the following option:

%– Allow post-pruning in combination (if post-prune enabled)

%post-pruning in combination=yes

post-pruning in combination=no

Unless the option for combination is "no combination", SMILES always gives statistics for the combination and statistics for the best single solution. In this way, SMILES tries to give the usually better accuracy results of the combination method jointly with the intelligible results of a single best solution.

C.3.7 Showing Several Solutions

Up to now we have always produced one best solution from the multi-tree (apart from the combination results that do not provide a comprehensible solution). However, there are some situations where it may be interesting to generate more than one solution. For this, we can determine the number of solutions output in the options file:

%–k best: how many solutions are generated in the multitree

k best number of solutions=3

The interest in obtaining more than one solution is maintained only if the solutions are different between them. In order to ensure that the solutions are quite different and still are good, SMILES has several "k-best" methods.

%–select k best method

%select k best=k-best less visited

%select k best=k-best less visited plus

select k best=k-best less visited then different components

%select k best=k-best different components

%select k best=k-best random

The previous methods are based on two main ideas: to avoid the use of the same branches that have been output for other solution (this is possible if several splits have been opened for an OR node), and secondly to avoid the use of the same partitions and attributes (herewith called components) in several solutions. In other words, these two ideas try to avoid repeated parts of solutions. For the second idea, a component matrix is used.

A component matrix describes in a very concise way which components take part in a decision tree. For instance, if we have 3 possible partitions and 4 attributes then a component matrix could be like this:

	$At0$	$At1$	$At2$	$At3$
$Part1$	1.5	0	0.5	0.5
$Part2$	0	0	0	0
$Part3$	0	0	0	0

A component matrix ignores the values of the conditions. For instance, the previous matrix could represent the following tree:

f(X0, X1, X2, X3) = R :- X0=overcast.
f(X0, X1, X2, X3) = R :- X0=rain, X3=weak.
f(X0, X1, X2, X3) = R :- X0=rain, X3=strong.
f(X0, X1, X2, X3) = R :- X0=sunny, X2=high.
f(X0, X1, X2, X3) = R :- X0=sunny, X2=normal.

The values are obtained depending on the times an attribute is used and the depth at which it is used. For instance, 1.5 corresponds to 3/2 because the split with X0 has three children at depth 1 (this gives the value 2^1 in the denominator). The value 0.5 corresponds to 2/4 because the split with X3 is used in two splits and at depth 2 (this gives the value 2^2 in the denominator).

SMILES allows the user to see the component matrix of each solution, just by changing the corresponding option in the options file:

%–show component matrix of solutions
%show solutions components=no show
show solutions components=show

As we have said the component matrix is used for obtaining different solutions. For this reason, a *reference* component matrix has to be updated in order to reflect what kind of partitions and attributes have to be avoided. There are three different ways of how to generate this component matrix between several solutions:

%–select k best components generation
select k best components generation=component accumulate
%select k best components generation=component random generated from start
%select k best components generation=component random generated from second

Let us show an example of how SMILES produces several solutions. If we use the playtennis dataset, 100 open or-nodes in the multi-tree, the “component random generated from second options" and "3 k-best solutions", we obtain the following output:

****** SMILES v.2.3.1 (Release Date: 23-August-2002) ******
ftrain: samples/playt.train
ftest: samples/playt.test
fcost:
ftestcost:
Training Set: "samples/playt.train"
No. of Attributes: 4
Cardinality: 14 examples
Class: 0. Distribution: 9
Class: 1. Distribution: 5
Valid options.
Creating the multitree.
Learning begins...
Tree #100 learned of 100
MeanDepth of Second Solution Start Positions: 1.77778
The test set: "samples/playt.test" will be used to evaluate the results
Predicting and preparing statistics
Showing first the results of the combination method of all branches:
COMBINATION RESULTS:
Statistics over test set of length: 15:
Relative Accuracy: 1
AUC (example by example) in Hand & Till's way: 1
Filling new test probabilities of the leaves of the multi-tree with the Test set
15 examples done.
Let us select the 3 best trees.
From 3 seeked, 3 solutions have been found.
Showing their properties
SOLUTION 0: 5 rules
Component Matrix:
P\R C 0| C 1| C 2| C 3|
C 0 1.5| 0| 0.5| 0.5|
C 1 0| 0| 0| 0|
C 2 0| 0| 0| 0|
Statistics over test set of length: 15:
Relative Accuracy: 1
AUC (example by example) in Hand & Till's way: 1
Let us show the solution: 0

```
f(X0, X1, X2, X3) = R :- X0=overcast. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=high. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=normal. [class: yes]
SOLUTION 1: 10 rules
Component Matrix:
P\R C 0| C 1| C 2| C 3|
C 0 1.75| 0.75| 0| 1.75|
C 1 0| 0| 0| 0|
C 2 0| 0| 0| 0|
Statistics over test set of length: 15:
Relative Accuracy: 1
AUC (example by example) in Hand & Till's way: 1
Let us show the solution: 1
f(X0, X1, X2, X3) = R :- X0=overcast, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=hot, X3=weak. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=mild, X3=weak. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=cool, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=overcast, X3=strong. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=hot, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=mild, X3=strong. [class: yes]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=cool, X3=strong. [class: yes]
SOLUTION 2: 8 rules
Component Matrix:
P\R C 0| C 1| C 2| C 3|
C 01.125| 0|1.375|0.875|
C 1 0| 0| 0| 0|
C 2 0| 0| 0| 0|
Statistics over test set of length: 15:
Relative Accuracy: 1
AUC (example by example) in Hand & Till's way: 1
Let us show the solution: 2
f(X0, X1, X2, X3) = R :- X0=overcast, X2=high. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X2=high, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X2=high, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=high. [class: no]
```

f(X0, X1, X2, X3) = R :- X2=normal, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=overcast, X2=normal, X3=strong. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X2=normal, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=normal, X3=strong. [class: yes]
END OF RESULTS
Calculating mean differences between solutions
Mean Discrepancy: 0.133333
Mean Syntactic Euclidean Distance: 1.49475
Mean Syntactic Manhattan Distance: 2.66667
Mean Accuracy: 0.933333
Maximum Accuracy: 1
Test dataset destroyed successfully
The multitree has been destroyed successfully
====> Time used (for learning): 0.03 secs.
****** Smooth end of SMILES execution ******

Three different solutions have been obtained, which have different components. At the end of the results, SMILES shows some statistics: *mean discrepancy* indicates the mean semantic difference between the solutions (i.e. the percentage of examples in which their predictions differ), two different syntactic distances which measure the difference in components between the solutions, the mean and maximum accuracy from all the solutions.

C.3.8 Validation Set and Cross-validation

A validation set is a subset of the training set that is not used for constructing the model, but for selecting between or combining the constructed models. A typical use of validation sets is the technique known as cross-validation.

In our case, if the training set is large enough we can "reserve" some part of it and use just a portion of it for training (we will call it subtraining set). This is especially useful with our multi-tree paradigm, since we can grow the multi-tree using the subtraining set as always, and then use the remaining dataset (the validation set) to select a good solution or combining between different solutions.

At the present moment, this option can be selected as follows:

%–sample training: whether or not a subset of the trainset is to be used
%sample training set=no sample training set
sample training set=sample training set

Note that if the "cross coverage" option is selected as the option for the best tree selection or "cross coverage combination" is selected, then the sample training must be activated.

Finally, the proportion (from 0 to 1) of the training set that is devoted for learning is specified through the following option:

%–sample training set portion: the portion of the trainset to be sampled
sample training set proportion=0.05

In the previous case, 5% of the data would be used for training and 95% of the data would be used for validation.

The use of a validation test can also be quite useful when the number of examples is too large to be handled by SMILES with a reasonable consumption of resources. Note that the sample is random and the rest is used for selection/combination. Consequently, this is better than making a manual sample before feeding the data to SMILES.

Another important feature of SMILES is *cross-validation*. Cross-validation is a powerful method to evaluate the quality of a classifier in a more reliable way. Cross-validation is based on the idea of automatically splitting the whole dataset into two parts: training set and test set. However, this split can be done randomly several times and obtain the average of all the results. This can give a much more accurate result than just evaluating one split (either manually done into training set file and test set file, or automatically).

In order to do cross-validation, the system should be supplied with a training set that contains all the data. In the smiles distribution, this kind of files can be found in the samples directory with names such as "/samples/monk2.all".

The options must be chosen in the following way to do cross-validation:

%–sample training: whether or not using a subset of the training set
sample training set=no sample training set
%sample training set=sample training set
%
%–sample training set portion: the proportion of the training set to be sampled
sample training set proportion=0.50
%
%– cross validation: use a different test set file for results or split the training set
%cross validation=use separate test set
cross validation=cross validation
%cross validation=kfold cross validation
%cross validation=repeated kfold cross validation
%
%– how many times (if cross validation) the split has to be done
k fold of cross validation=10
%

%– how many times (if cross validation) we repeat the experiment

repeat kfold=1

Note that the "sample training" option must not be enabled. The "sample training set portion", however, is necessary and tells SMILES that the input data file is going to be split into two datasets (training and test) of equal size (50% and 50%). The next option is the key one: "cross-validation" that must be selected "cross validation", in order to make SMILES not expect the test set, and let it extract its test set. Finally, the last option "k fold of cross validation" selects the times that different splits have to be done. Note that each fold performs a different and random split of the initial dataset into two different sets (training set and dataset) and learns the multi-tree, with all the associated process.

For instance, if we use the file "samples/monks2.all" with MSE split and 100 trees, then we would have 10 runs of the algorithm. At the end, a summarised listing of the 10 runs is included and, ultimately, the mean and standard deviation of these 10 runs are computed, giving a portrait as follows:

Mean Results:

N. of susp. nodes explored : 100 +/- 0

Solutions in the Multitree : 6.31864e+09 +/- 7.02266e+09

Results for 1st Solution:

Accuracy of 1st Solution : 0.701667 +/- 0.0541203

AUC (by nodes) of 1st Sol : 0.710585 +/- 0.0698885

Mean # Rules : 286.9

Results for Combination:

Accuracy of Combination : 0.768333 +/- 0.0677914

AUC of Combination : 0.768977 +/- 0.0550205

Results for Best Solution:

Accuracy of Best : 0.693333 +/- 0.073367

Mean # Rules : 275.2

Accuracy class 0 : 0.756506 +/- 0.0753576

Accuracy class 1 : 0.566578 +/- 0.0963091

AUC by Hand : 0.680732 +/- 0.0821538

MSE : 0.719583 +/- 0.0714988

LogLoss : 0.740103 +/- 0.0827415

Time Used : 0.273 +/- 0.0498999

Mean results are computed for a lot of measures. First some statistics on the multi-tree are shown. Then, the mean results of the first solution are shown: its accuracy and its AUC (Area Under the ROC Curve), as well as the number of rules. Secondly, the same results for combination are shown (except, logically, the number of rules). Finally, for the best solution further measures are shown. The AUC, MSE and LogLoss measures will be discussed later. AT the end, the mean time used for each iteration is shown.

C.3.9 k-fold Cross-Validation and repeated k-fold Cross-Validation

K-fold Cross-Validation is the usual way to use cross-validation. The idea is to use all the possible combinations of a partition. For instance if k=10, we can partition the dataset into ten parts. Then, we can select ten different combinations of 9 parts for the training set and 1 part for the test set.

The way to use k-fold Cross-Validation in SMILES is quite simple:

%–sample training: whether or not a subset of the training set is to be used
%sample training set=sample training set
%
%–sample training set portion: the proportion of the training set to be sampled
sample training set proportion=0.90
%
%– cross validation: use a different test set file for results or split the training set
cross validation=repeated kfold cross validation
%
%– how many times (if cross validation) the split has to be done
k fold of cross validation=10
%
%– how many times (if cross validation) we repeat the experiment
repeat kfold=1

The "sample training set proportion" (lets call it p) tells which proportion is used for training, as always. The difference is that now, each iteration a different subset of the *same* partition is used. Note that the meaning of any combination of p and k is clear when $(1-p)*k <= 1$. Other combinations are implementation dependent.

Finally, in some cases 10-fold validation could not give a quite reliable information about the quality of a hypothesis. A good idea is to augment k, even to match the number of examples, known as all-to-1 cross-validation.

Another way to augment reliability of the means computed by SMILES is the use of repeated experiments. This option is called "repeated kfold cross validation". For instance, if we use the previous options with these modifications:

%– cross validation: use a different test set file for results or split the training set
cross validation=repeated kfold cross validation
%– how many times (if cross validation) we repeat the experiment

repeat kfold=20

Then the experiment will be repeated 20 times, i.e. we make 20 partitions and exploit each combination of each partition. In the previous case, we would have 20x10= 200 iterations, from which the means are computed. For instance, the following output shows the results for monks2 with MSE split and 100 trees, and 10x10= 100 runs of the algorithm.

Mean Results:

N. of susp. nodes explored : 100 +/- 0

Solutions in the Multitree : 8.07935e+10 +/- 4.18201e+11

Results for 1st Solution:

Accuracy of 1st Solution : 0.683 +/- 0.0511364

AUC (by nodes) of 1st Sol : 0.666375 +/- 0.0615097

Mean # Rules : 282.79

Results for Combination:

Accuracy of Combination : 0.750167 +/- 0.0583391

AUC of Combination : 0.741057 +/- 0.0611061

Results for Best Solution:

Accuracy of Best : 0.692 +/- 0.0524634

Mean # Rules : 270.71

Accuracy class 0 : 0.755819 +/- 0.0685639

Accuracy class 1 : 0.576554 +/- 0.0986817

AUC by Hand : 0.669107 +/- 0.0568279

MSE : 0.712333 +/- 0.0493715

LogLoss : 0.743794 +/- 0.0519433

Time Used : 0.2756 +/- 0.0591577

As we can see the deviations are now lower than for just 10 cross-validation.

When small datasets are used or low proportions are used for cross-validation, there is a higher possibility than in one partition of the dataset one class wouldn't appear in any example of the validation test dataset. In this case, the evaluation results would not be as accurate. In order to avoid this there is simple (although inefficient) way: if a partition leaves the validation test dataset without examples of any class, than the partition is repeated. This can be enabled through the following options:

%–allow (if cross validation) a test dataset with one class without examples

%allow test without one class=yes

allow test without one class=no

C.3.10 Expected Error and Smoothing Options

The current system includes several ways to compute the expected error. This value can be used or not depending on other options, especially if expected error pre-pruning is active or some other criterion is based on it (split criterion, best tree criterion).

%–Expected error method: several ways to compute expected error

%expected error method=no compute

expected error method=relative frequency with majority class

%expected error method=relative frequency with frequency probability

%expected error method=cost with minimum class

%expected error method=cost with frequency probability

%expected error method=cost with cost probability

The relative frequency used for computing the expected error can be smoothed. Different smoothing criteria can be selected:

smoothing method=no smoothing

%smoothing method=laplace

%smoothing method=k-estimate

%smoothing method=m-estimate

%smoothing method=m-estimate uniform

Smoothing can also be used for assigning the majority class when using cost information, as we will see in the next subsection, and it can be used in splitting criteria, as we describe next.

The cardinalities of each node in a split are taken into account in some splitting criteria. The derived probabilities can be smoothed in different ways:

%–frequency error smoothing:use smoothing or not for node probabilities in a split

node smoothing method=no smoothing

%node smoothing method=laplace

%node smoothing method=k-estimate

%node smoothing method=m-estimate

%node smoothing method=m-estimate uniform

According to our experiments, this smoothing does not improve in general.

Similarly, many splitting criteria are based on the probabilities of the nodes under the split. This probability can be computed from the frequency directly or it can be computed in other more sophisticated ways. These are the current options:

%–Probability in splitting criteria

probability in splitting criteria=from frequency no smoothing

%probability in splitting criteria=from frequency smoothing

%probability in splitting criteria=from costs

%probability in splitting criteria=from frequency with stratification

%probability in splitting criteria=from frequency with stratification no smoothing
None of these options is relevant to the Descriptive MDL Splitting Criterion, because this method does not use probabilities. For the Local ROC Splitting Criterion only the two first probabilities (with and without smoothing) can be used and are effective.

Frequency smoothing is especially recommended for CART and DKM splitting criteria. Both of them do not work well for more than two classes. For more than two classes, if you want to use a similar criterion, use MGINI.

C.3.11 Cost-sensitive and ROC Analysis Features

Accuracy (or error), i.e., percentage of instances that are correctly classified (respectively incorrectly classified) has been traditionally used as a measure of the quality of classifiers. However, in most situations, not every misclassification has the same consequences. In fact, it is usually the case that misclassifications of minority classes into majority classes (e.g. predicting that a system is safe when it is not) have greater costs than misclassifications of majority classes into minority classes (e.g. predicting that a system is not safe when it actually is). Obviously, the costs of each misclassification are problem dependent, but it is almost never the case that they would be uniform for a single problem. Consequently, accuracy is not generally the best way to evaluate the quality of a classifier or a learning algorithm.

Although there can be other kinds of cost associated with predictions (e.g. test cost that we will address later on), the most relevant ones are misclassification costs, i.e., the cost of classifying an instance of class a into class b. All these misclassification costs for a specific problem can be arranged in a c-dimensional matrix, with c being the number of classes. This matrix is called a cost matrix (see Section 5.2).

The use of cost matrices for the generation of classifiers that minimise the resulting prediction cost instead of the prediction error has been incorporated in a few aspects of a few learning systems by changing some criteria or measures used by these methods [Pazzani *et al.*, 1994; Bradford *et al.*, 1998; Kukar and Kononenko, 1998]. Nonetheless, it is also common to use a learning system that is not cost-sensitive and to modify the class distribution of the training data set to obtain a classifier that adjusts itself to a specific cost matrix and the class distribution of the test set if known [Lavrac *et al.*, 1996; Fan *et al.*, 1999; Domingos, 1999].

However, a change of class distribution is usually done by stratification (or rebalancing), i.e., either by under-sampling or by over-sampling. Stratification presents some problems though (lost of data or redundant data).

The usefulness of cost-sensitive learning does not only apply when the cost matrix is known a priori. If the cost matrix is not known, one or many classifiers can be generated in order to behave well in the widest range of circumstances or contexts as possible. The Receiver Operating Characteristic (ROC) analysis 5.2.2 provides tools to select a set of classifiers that would behave optimally and reject some other useless classifiers.

From the cost matrix and the confusion matrix it is very easy to compute the cost of a classifier for a given dataset, just as the 1 by 1 matrix product, given a Resulting Matrix:

$$R(i,j) = M(i,j) \quad \cdot \quad C(i,j)$$

Our system SMILES incorporates many features that can handle cost information, either given as class weights or as a cost matrix.

The first thing to tell SMILES is how the cost matrix is going to be specified. This once again can be done through the option file:

%– weights method:how the cost matrix is constructed
weight method=no costs
%weight method=uniform weights
%weight method=inverse frequency weights
%weight method=weights from file
%weight method=costs from matrix

The first three options do not need any additional information. The next two require some information, either a line in the training set file or a separate cost matrix file.

The "no costs" option is the default option and it is equivalent to the use of "uniform weights", which means a matrix with all equal costs. The difference is just in efficiency, since the first option does not force SMILES to do any cost calculation.

The third option generates class weights according to the class distribution. For instance, if a training set of three classes has distribution (500, 2000, 2500) , then the following weights are generated (c/500, c/2000, c/2500) where c is a parameter that can be modified through program (hardwired option). This inverse class distribution assigned weights try to give more relevance to the classes with less cardinality in order to "compensate" the dataset.

As we have discussed before, accuracy is frequently a much too simplified measure of the quality of a classifier. For instance, given a dataset whose distribution of classes is (p_a= 0.85, p_b= 0.1, p_c= 0.05), i.e., most of the examples are of the class 'a', a simple classifier predicting everything into class 'a' would have 85% of accuracy.

The weight can also be read from the training file if the option "weights from file" is chosen. Then SMILES would look for a line as follows:

!WEIGHTS:yes=1,no=2

as we discussed in Section C.3.2.

For this two latter options, we have talked about weights, whereas we talked about costs elsewhere. How is a weight vector converted into a cost matrix?

For instance, from this weight vector:

	a	b	c
$Weights$	3	5	2

If we could use over-sampling with these weights, then we would have that the frequency of class 'a' would be multiplied by $3/2$, and the frequency of class 'b' by $5/2$, by conveniently duplicating some of the examples.

It is easy to show that the corresponding cost matrix to this over-sampling would be:

		Actual		
		a	b	c
	a	-3	5	2
Predicted	b	3	-5	2
	c	3	5	-2

Moreover, for two classes it is easy to show that the resulting assignments will be exactly the same that if we put 0s on the diagonal.

Finally, the easiest way to create a matrix is to set the "costs from matrix" option. In this case, SMILES will look for a cost file specified in the command line (the third file), as the usage shows:

USAGE:

./smiles file.train [file.test] [file.cost] [file.testcost]

Otherwise this file will be ignored. If we do not want to specify the test set, we just place the symbol "-". For instance, the following command line, would just look for a train set file and a cost file.

./smiles samples/liver.all - liver.cost

The format of the file is just a list of real numbers separated by commas. The last one must also have a comma, such in the following example:

```
% Matrix 4x4 for cars problem
0, 3.2, 1.1, 4.3,
2.5, 0, 10.4, 8.2,
3.2, 17.1, 0, 0.1,
2.3, 8.2, 4.1, 0,
```

As always, lines beginning with '%' are ignored.

Now that we know how to construct the cost matrix in several ways, what can we do with it? The first and most effective thing to do is to change the way in which the classes are assigned to each node. Instead of assigning the majority class (the default option), i.e. the most frequent class, we can label a node with the class that *minimises the cost.* The possible ways are as follows:

%–class selection method: how the class of a leaf is assigned
class selection method=majority class
%class selection method=minimum cost selection
%class selection method=minimum cost class without smoothing
%class selection method=stratification class
%class selection method=stratification without smoothing

The "minimum cost selection" is the option that assigns the class taking into account the costs, i.e., if we have a leaf vector $V(i)$ then we look for the class i such that:

$$\text{Assigned Class} = \arg\min_i \sum_j C(i,j) \cdot V(j)$$

For instance, if a leaf node has $V = \{20,\ 10,\ 22\}$ and we have the following cost matrix:

Actual

		a	b	c
	a	0	10	5
Predicted	b	1	0	2
	c	5	3	0

If we assign class a to the node we have a cost of $20 \cdot 0 + 10 \cdot 10 + 22 \cdot 5 = 210$. If we assign class b to the node we have a cost of $20 \cdot 1 + 10 \cdot 0 + 22 \cdot 2 = 64$. If we assign class c to the node we have a cost of $20 \cdot 5 + 10 \cdot 3 + 22 \cdot 0 = 130$. It turns out that despite that c is the majority class, b is the less costly class.

Note that when pruning is not active all the leaf nodes are pure and hence, the options "minimum cost selection" and "minimum cost class without smoothing" are equal to the majority class provided the cost matrix is normalised (all values are positive and only 0s in the diagonal). The stratification methods are usually worse because they use a simplified version of the cost matrix. They are useful for comparing with other methods and are not recommended.

The class selection method is the most important option in the sense that it has a very effective impact on cost minimisation. Other less effective (or even with no provable good effects) are based on modifying some criteria used during learning.

One first idea that turned out to be poorly successful was to modify the splitting criteria taking costs into account. Since most of them are based on probabilities, the idea was to modify these probabilities taking cost into account. Although we saw the probability in the splitting criteria before, the three last possibilities can now be understood as ways to compute this probability based on expected costs instead of expected frequencies.

%–Probability in splitting criteria
probability in splitting criteria=from frequency no smoothing

%probability in splitting criteria=from frequency smoothing

%probability in splitting criteria=from costs

%probability in splitting criteria=from frequency with stratification

%probability in splitting criteria=from frequency with stratification with stratification

In the case that the third one is chosen ("from costs"), then there are two ways of deriving this probability, that can be selected through the option file:

%–cost derived probability method

cost derived probability=direct

%cost derived probability=with smoothing

There are also cost-sensitive options in how to compute the expected error, which would turn into an expected cost. This would be used in any other option that uses expected error. Costs can also be used in "multitree solution combination" options.

Finally, there are some facilities related to ROC analysis. These are possible for problems with 2 classes (in the following sections we will see measures that are applicable for more than 2 classes).

First of all we can compute and show the ROC points obtained by the optimal assignments of a classifier. Note that these points are obtained from a single "soft" classifier, not as usual, when we obtain a ROC curve from many classifiers. These points represent different assignments of the same trees, that, in fact, give different classifiers:

%–compute ROC points

compute ROC points=no

%compute ROC points=yes

%

%– show ROC points

show ROC points=no

%show ROC points=yes

From these points we can draw and compute the area by using these options:

%– compute ROC area

compute ROC area=no

%compute ROC area=yes

%

%– generate ROC curve file

%generate ROC curve file=no

generate ROC curve file=yes

The outputs of each solution are generated into a postscript file called "ROCtstN.ps" for the training set, "ROCtotN.ps" when the training set is used for ordering the nodes and the test set to compute the leaves probabilities, and "ROCtstN.ps" for the test set, where N is the number of the solution. If there is only one solution, then N is just 0.

These ROC features are thoroughly explained in Section 11.

C.3.12 AUC Evaluation

We have just described that different curves can be output: with respect to the training set, using the training set for the ordering and the test set to compute the leaves probabilities and the entire curve with the test set. From these curves we can compute the area under the ROC curve, which is very useful to estimate the quality of a classifier. In particular the AUC measures used and output by SMILES are:

- **AUC0:** the curve is constructed with the order derived from the training set and with the node distributions given by the training set. If pruning is not active, it is usually 1. Consequently this measure is not much too informative.
- **AUC1:** the curve is constructed with the order derived from the test set and with the node distributions given by the test set. This way to evaluate a classifier seems cheating, because the order is derived from the test, which could not be performed in general. What this measure tells is that if this value is high, there will be good labellings of the tree that could obtain high accuracies for different cost matrices, but it does not tell that we are necessarily going to be able to use these optimal labellings.
- **AUC2:** the curve is constructed with the order derived from the training set and with the node distributions given by the test set. This is a very useful measure. What this measure tells is that if this value is high, there will be good labellings of the tree that could obtain high accuracies for different cost matrices, and since we are using the order from the training set, we can know that order and we will be able to *use* these optimal labellings.
- **AUC4 (1P-AUC)**: it is a simplification of AUC2 that only uses one point for computing the curve. The interest of this measure is that can be used for more than 2 classes (currently it is not implemented for more than 2 classes).

All the AUC measures have singular values when FPR=0 or TPR=0. We have assumed the following assignments. If FPR=0 then AUC=TPR, and if TPR=0 then AUC=FPR.

Theoretically, AUC1<AUC0, AUC2<AUC1 and AUC4<AUC1. These measures are shown with cross-validation when ROC options are active (option "compute ROC points=yes"). For monks2, for instance, this could be a possible result (only the excerpt for the best solution is shown):

Results for Best Solution:
Accuracy of Best : 0.692 +/- 0.0524634
Mean # Rules : 270.71
Accuracy class 0 : 0.755819 +/- 0.0685639
Accuracy class 1 : 0.576554 +/- 0.0986817
AUC0 (train) : 0.185172 +/- 0.014343
AUC1 (test) : 0.991017 +/- 0.00788894
AUC2 (train-order + test) : 0.669008 +/- 0.0594674
AUC4 (train-1-lab + test) : 0.666187 +/- 0.0528094
AUC by Hand : 0.669107 +/- 0.0568279
MSE : 0.712333 +/- 0.0493715
LogLoss : 0.743794 +/- 0.0519433

If possible, AUC2 should be used instead of accuracy to evaluate classifiers when class distributions or costs might change when the model were to be applied. This is the one that is shown by default (computed in Hand and Till's way) and shown as "AUC by Hand". AUC by Hand and AUC2 are not exactly the same because of the ordering functions. When two or more nodes have the same ratio, then their precise order is not defined, and hence small variations can occur if both methods order them differently (this is solved since version 2.1.7). In the next section we better explain how AUC2 is computed in Hand and Till's way.

C.3.13 Multi-class AUC Evaluation

The problem of previous AUC measures and ROC analysis is that they are only applicable to problems with two classes and not valid for multi-class problems. Only the 1-point AUC measure is extensible. However this measure does not take into account the possible labelling that can be done in a decision tree, turning it into a soft classifier.

Fortunately there are some extensions and approximations of the AUC measure for more than two classes (see Section 5.2.2 and Section 11.5.1). The first one is Hand and Till's M Function (that we will call AUCH), which is the most popular extension, and the other two (MSE and log-loss) are traditional measures adapted for this purpose. All of them try to consider that not all the errors have the same consequences and that more compensated solutions are preferable from those that would be selected by using accuracy.

These measures have been adapted as splitting criteria, as we have discussed in Section 11.5.1. The configuration of SMILES system in order to use these criteria is described in Section C.3.4.

C.3.14 Test Cost

Apart from misclassification costs, there is another kind of costs that can be extremely important in some applications, especially in medical diagnosis, the test costs, as described in Section 5.2.4.

SMILES provides tools to compute the test cost and to use it in a reasonable way, in order to obtain trees with lower cost or to select from a pool of trees (the multi-tree) the tree with lower test cost. Let us review the facilities that SMILES offers in this regard.

The first obvious thing to be done is to allow SMILES to read the test cost information. For this, there is an optional additional file that can be specified through the command line:

USAGE:

./smiles file.train [file.test] [file.cost] [file.testcost]

If we do not want to specify neither a test set nor a cost matrix, we just place the symbol "-". For instance, the following command line would just look for a train set file and a test cost file.

./smiles samples/liver.all - - liver.testcost

The format of the file is just a list of real numbers separated by commas. The last one must also have a comma, such in the following example:

% Test costs from UCI liver bupa problem

7.27, 7.27, 7.27, 7.27, 9.86, 1

As always, lines beginning with '%' are ignored. However, if this file is specified in the command line we must also enable one option in the options file, that tells SMILES to read test costs from files.

%–test cost method: how the vector of attribute test costs is constructed

%test cost method=no test costs

%test cost method=uniform test costs

test cost method=test costs from file

SMILES shows the information has read. For instance, for the following command line:

./smiles samples/liver.all - - liver.testcost

SMILES outputs the following after reading the training set:

...

Test Cost Vector to be used:

Argument 0: 7.27

Argument 1: 7.27

Argument 2: 7.27

Argument 3: 7.27

Argument 4: 9.86

Argument 5: 1

...

As we can see there are two other options: "no test costs" which does not take this kind of cost into account and "uniform test costs", which, when we do not know any information about the test costs, we can assume that these are uniform, and SMILES will look for trees that minimise the average number of tests per example, in a similar way as is made in ROC analysis. In other words, if AUC can be computed when the real cost matrix is not know, the uniform test cost vector can be used when the test cost vector is not known.

Now that we know how to read a particular test cost vector or to assume a uniform one, let us explain how SMILES can take advantage of it.

When constructing each tree, a splitting criterion is used to assign different degrees of optimality to each split. This optimality can be modified taking test cost into account. The test cost is computed using the cardinality of each node, i.e., multiplying the examples that fall into each branch by the cost of the tests (attributes) used until that node. Once the test cost is computed, there are three ways of treating test cost information for modifying splitting criteria:

%–test cost use: how the vector of attribute test costs is used

test cost use=test costs no use

%test cost use=test costs linear plus1 without repetition

%test cost use=test costs linear plus1 with repetition

The first option logically makes no use of this information. The second and third options use the formula introduced in Section 11.7 to modify the existing splitting optimality (Opt) according the test cost of attributes.

The difference between the two last options (without or with repetition) is that numerical attributes can be used more than once in the same branch. Consider e.g. a condition $X < 3.2$; this could be followed by an additional partition below on the same attribute in the same branch such as $X < 1.3$ and $X >= 1.3$. It is not sensible to compute the test cost of attribute X more than once. For this reason, it is more accurate the use of the option "without repetition" that only takes into account this cost once. This gives an exact measurement of test cost.

This modification provides a way to construct trees and multi-trees that have lower average test costs per example. Obviously, if test cost is given too much importance then accuracy can be affected.

Despite the effect and good results of the previous approach, it is not much wise to use a test cost sensitive criterion and then select the best tree using non-test cost sensitive criteria. For this, SMILES also includes two methods for extracting the best solution from the tree taking test costs into account.

%–multitree: best tree selection criterion

%multitree best tree criterion=test cost best

%multitree best tree criterion=occam and test cost best

Apart from the "occam best" and others, SMILES provides "test cost best" and "occam and test cost best". The first method selects the tree with the lowest test cost (repeated use of the same attribute in the same branch are discarded). The "occam and test cost best" combines "occam best" and "test cost best" through the use of a weight factor. The weight of each is determined through the following formula:

$$test\cos t^{\alpha} \cdot numrules^{(1-\alpha)}$$

Where α is a factor modifiable by program by varying the new option value $TestCostRelevanceInSelectBest$ that must be from 0 to 1.

Finally, test cost vectors can be used to give more relevance to some attributes than others, e.g. when the user thinks that some attributes can be more useful or comprehensible than others.

Let us see an example with the monks2 dataset. Its test costs are given in the file "monks.testcost":

% Test costs for monks problems

1.0, 0.0, 10.0, 5.0, 0.0, 100.0

If we run SMILES with MSE splitting criterion, and a multi-tree with 100 open nodes and with 10x10 cross validation, but without using test cost, we have:

Mean Results:

N. of susp. nodes explored : 100 +/- 0

Solutions in the Multitree : 8.07935e+10 +/- 4.18201e+11

Results for 1st Solution:

Accuracy of 1st Solution : 0.683 +/- 0.0511364

AUC (by nodes) of 1st Sol : 0.666375 +/- 0.0615097

Mean # Rules : 282.79

Test Cost per Example : 96.038 +/- 9.27396

Results for Combination:

Accuracy of Combination : 0.750167 +/- 0.0583391

AUC of Combination : 0.741057 +/- 0.0611061

Results for Best Solution:

Accuracy of Best : 0.692 +/- 0.0524634

Mean # Rules : 270.71

Accuracy class 0 : 0.755819 +/- 0.0685639

Accuracy class 1 : 0.576554 +/- 0.0986817

AUC by Hand : 0.669107 +/- 0.0568279

MSE : 0.712333 +/- 0.0493715

LogLoss : 0.743794 +/- 0.0519433

Test Cost per Example : 103.333 +/- 8.65866

Time Used : 0.2765 +/- 0.0581599

As we can see now, we have "Test Cost per Example" information for the first solution and for the Best Solution. For the first solution we have 96.038 test cost units, and for the best solution we have 103.333.

Now we can see the results if we enable

test cost use=test costs linear plus1 without repetition

Then we have:

Mean Results:

N. of susp. nodes explored : 100 +/- 0

Solutions in the Multitree : 1.2661e+14 +/- 6.4715e+14

Results for 1st Solution:

Accuracy of 1st Solution : 0.687667 +/- 0.0535297

AUC (by nodes) of 1st Sol : 0.607398 +/- 0.0676029

Mean # Rules : 324.36

Test Cost per Example : 46.2625 +/- 6.58806

Results for Combination:

Accuracy of Combination : 0.736667 +/- 0.0566558

AUC of Combination : 0.671598 +/- 0.0699609

Results for Best Solution:

Accuracy of Best : 0.682667 +/- 0.0624922

Mean # Rules : 309.23

Accuracy class 0 : 0.74696 +/- 0.0755163

Accuracy class 1 : 0.566277 +/- 0.0993084

AUC by Hand : 0.648622 +/- 0.0827158

MSE : 0.689375 +/- 0.0669407

LogLoss : 0.8372 +/- 0.0476209

Test Cost per Example : 77.3888 +/- 17.9072

Time Used : 0.2716 +/- 0.0625344

We can see that test cost is dramatically reduced, with a slight loss in accuracy. We see that the best solution has greater cost than the first solution. This is so because the best solution is defined "Occam best", in order to obtain the shortest one, not the one with lowest test cost.

We can even reduce the test cost of the best solution by using the option:

multitree best tree criterion=test cost best

And now we have:

Mean Results:

N. of susp. nodes explored : 100 +/- 0

Solutions in the Multitree : 1.2661e+14 +/- 6.4715e+14

Results for 1st Solution:
Accuracy of 1st Solution : 0.687667 +/- 0.0535297
AUC (by nodes) of 1st Sol : 0.607398 +/- 0.0676029
Mean # Rules : 324.36
Test Cost per Example : 46.2625 +/- 6.58806
Results for Combination:
Accuracy of Combination : 0.736667 +/- 0.0566558
AUC of Combination : 0.671598 +/- 0.0699609
Results for Best Solution:
Accuracy of Best : 0.685667 +/- 0.0546399
Mean # Rules : 323.39
Accuracy class 0 : 0.753444 +/- 0.0629829
Accuracy class 1 : 0.561628 +/- 0.0993743
AUC by Hand : 0.610403 +/- 0.0687935
MSE : 0.685792 +/- 0.0541103
LogLoss : 0.845749 +/- 0.0451748
Test Cost per Example : 46.6472 +/- 6.5868
Time Used : 0.2716 +/- 0.0648654

We see that the test cost for the best solution is now significantly lower than before. Unlike this example, in general, with these options it is usually lower than the one for the first solution.

C.3.15 Archetype Solution

The use of a multi-tree allows combination to obtain more accurate results. However, comprehensibility is lost with combined hypothesis. Moreover, combined solutions explore many alternatives and, consequently, they have an extremely high test cost. Consequently they are not useful when test costs is to be taken into account. This makes combination methods useless for many applications, such as medicine, where tests are expensive, and may also be risky and painful. It is not sensible to make all the existing tests to a patient, because a machine learning algorithm requires a multiplicity of different solutions.

But we have described before that a single solution can be extracted from the pool of solutions (with several best single solution methods). However, the solution is usually not as good (by far) as the combined solution.

An original idea introduced in SMILES is the notion of "archetype" or "representative" (see Section 10.4). Combined solutions are usually much better than a single solution. Combined solutions have a behaviour that is different from any single solution, but, in many cases, one or more solutions are quite closer (in a semantic way) to the combined solution. Why not choosing the single solution that is semantically closer to the combined solution? This is the idea of the *archetype* or *representative* of a group: the individual that best represents the group.

Another original idea of SMILES is that it does not require an additional dataset to compute which is the solution that is most similar to the combined one. SMILES can construct an invented dataset for this. The first thing we must tell SMILES is to extract this archetype to the use of an invented dataset. This is done through the "combination to single solution":

%– Combination to Single Solution Options

%combination to single solution method=no extraction

combination to single solution method=invented dataset

Once determined the construction of an invented dataset, we must tell the size of this invented dataset.

%– Length of the invented random dataset

invented dataset length=10000

The greater the invented dataset the better the estimation of the archetype will be (and it will be slower too). Finally, the last thing to specify is how the similarity is to be computed. For this SMILES provides three similarity functions to be applied to the invented dataset.

%– Similarity function used for the selection of a single solution from the combination

similarity method for combination to single=kappa

%similarity method for combination to single=kappa1

%similarity method for combination to single=qstat

According to our experiments, the best similarity methods are "kappa" (the κ measure) and "kappa1" (the θ measure).

Let us see an example. If we run "monks2" in SMILES with MSE splitting criterion, and a multi-tree with 100 open nodes and with 10x10 cross validation, and select the previous options, as well as the use of the test cost ("monks2.testcost"), the option that restricts archetype criteria to just similarity, i.e.:

%– Combination to Single Solution (Archetype) Use of Other Criteria

archetype similarity importance factor=1.0

archetype occam importance factor=0.0

archetype test cost importance factor=0.0

Then we would have:

Mean Results:

N. of susp. nodes explored : 100 +/- 0

Solutions in the Multitree : 4.12571e+15 +/- 3.88401e+16

Results for 1st Solution:

Accuracy of 1st Solution : 0.687667 +/- 0.0535297

AUC (by nodes) of 1st Sol : 0.607398 +/- 0.0676029
Mean # Rules : 324.36
Test Cost per Example : 46.2625 +/- 6.58806
Results for Combination:
Accuracy of Combination : 0.737833 +/- 0.0522155
AUC of Combination : 0.676091 +/- 0.0692213
Results for Archetype:
Accuracy of Archetype : 0.720333 +/- 0.0520392
AUC (by nodes) of Archetype: 0.670872 +/- 0.0742581
Mean # Rules : 321.89
Test Cost per Example : 63.6257 +/- 16.246
Results for Best Solution:
Accuracy of Best : 0.685167 +/- 0.0593121
Mean # Rules : 308.93
Accuracy class 0 : 0.75152 +/- 0.0752203
Accuracy class 1 : 0.564675 +/- 0.0977124
AUC by Hand : 0.648288 +/- 0.0775428
MSE : 0.692875 +/- 0.0626703
LogLoss : 0.83394 +/- 0.0513109
Test Cost per Example : 77.5215 +/- 17.3678
Time Used : 0.2738 +/- 0.0758358

As we can see the combination has an accuracy and AUC of 0.738 and 0.676 respectively, which are significantly higher than the accuracy and AUC of the First and Best Solutions. However, we have obtained an archetype whose accuracy and AUC are quite closer to the combination, and is a single solution.

Finally, we have seen several methods to extract a single solution from the multi-tree: Occam-best, Testcost-best, a combination of both, and, finally, we have just seen the semantic archetype. It makes sense to be able to obtain a single solution that combines the Occam criterion, Testcost criterion and the semantic criterion. SMILES allows this through three options:

%– Combination to Single Solution (Archetype) Use of Other Criteria
archetype similarity importance factor=15.0
archetype occam importance factor=4.0
archetype test cost importance factor=1.0

Considering these factors f1, f2 and f3, the exact formula for this combination is (as seen in Section 11.7):

$$similarity^{f1} \cdot \left({}^{1}\!/\!{}_{numrules^{f2}} \right) \cdot \left({}^{1}\!/\!{}_{test\cos t^{f3}} \right)$$

These factors affect how the Archetype is extracted. If all the factors except similarity are left to 0, then it is just a semantic extraction

But if these factors are used, then we can extract an archetype taking into account its length and its test cost too.

This permits the user to extract the best single solution according to the user's relevance: comprehensibility and shortness (Occam), accuracy and AUC (similarity) and test cost (test cost factor).

For the previous example (monks2), if we use weights 15, 4 and 1, now we would have:

Mean Results:
N. of susp. nodes explored : 100 +/- 0
Solutions in the Multitree : 4.12571e+15 +/- 3.88401e+16
Results for 1st Solution:
Accuracy of 1st Solution : 0.687667 +/- 0.0535297
AUC (by nodes) of 1st Sol : 0.607398 +/- 0.0676029
Mean # Rules : 324.36
Test Cost per Example : 46.2625 +/- 6.58806
Results for Combination:
Accuracy of Combination : 0.737833 +/- 0.0522155
AUC of Combination : 0.676091 +/- 0.0692213
Results for Archetype:
Accuracy of Archetype : 0.715833 +/- 0.0527778
AUC (by nodes) of Archetype: 0.647856 +/- 0.0750104
Mean # Rules : 318.3
Test Cost per Example : 57.2522 +/- 9.0557
Results for Best Solution:
Accuracy of Best : 0.685167 +/- 0.0593121
Mean # Rules : 308.93
Accuracy class 0 : 0.75152 +/- 0.0752203
Accuracy class 1 : 0.564675 +/- 0.0977124
AUC by Hand : 0.648288 +/- 0.0775428
MSE : 0.692875 +/- 0.0626703
LogLoss : 0.83394 +/- 0.0513109
Test Cost per Example : 77.5215 +/- 17.3678
Time Used : 0.2743 +/- 0.0758555

Now we can see that the archetype has lost some similarity to the combination (and consequently accuracy and AUC are reduced) but the number of rules and the test cost is also reduced. That is what we wanted, a single solution that takes into account the similarity with the combination, a short number of rules and a low test cost.

The weights assigned to each criteria are problem dependent, but according to the usual absolute amounts of each factor, the values 15, 4 and 1 represent a compromise of the three criteria. The user can change these values in order to give more relevance to each desired characteristic of the model.

C.3.16 Other Facilities

There are other options mainly related to output. They are self-explained by their description in the options file.

```
%–output: syntax to show the rules:
show rules mode=functional-logic
%
%–output class dist.: show the trainset examples falling in each rule.
show class distribution=no
%show class distribution=yes (not implemented)
%
%–show all multitree rules
show all multitree rules=no show
%show all multitree rules=show
%show all multitree rules=to file
%
%–show all k-best solutions
show all k-best solutions=no show
%show all k-best solutions=show
%show all k-best solutions=to file
%
%–show component matrix of solutions
show solutions components=no show
%show solutions components=show
%
%–show confusion matrix
%show confusion matrix=no show
%show confusion matrix=show only if costs
show confusion matrix=show
%
%–how to show statistics
%show statistics=absolute statistics
%show statistics=relative statistics
%show statistics=both statistics
show statistics=just accuracy
%
%–show number of possible solutions in the multitree
```

show number of multitree possible solutions=yes

%show number of multitree possible solutions=no

At the present version of SMILES there are no other options (from the ones already explained) that could be modified through the options file. There are other other hardwired options for programmers or expert users.