Solving Classification Problems through Automatic Programming

Stig-Erland Hansen
stig.e.hansen@hiof.no

Master Thesis

Department of Computer Science
Østfold University College

July 3, 2007
Halden, Norway

Abstract

This thesis presents two approaches to how automatic programming can be applied to the well-studied field of classification through the use of the automatic programming system, Automatic Design of Algorithms Through Evolution (ADATE).

First, we evaluate whether the inductive bias of ADATE is better suited in general to induce classifiers than the inductive bias of other classification algorithms specifically designed for this task. Our results show that ADATE is competitive with state-of-the-art classification algorithms when applied to a set of classification problems from the UCI machine learning repository, although the execution time of ADATE is order of magnitudes higher than the combined execution time of the 32 other classification algorithms tested. In a separate paper given in appendix A, we use most of these classification algorithms to classify infrasound events pre-processed with wavelet transforms.

Second, we investigate the feasibility of improving existing classification algorithms through automatic programming by using ADATE to rewrite the code for the so-called error based pruning algorithm that is an important part of Quinlan's C4.5 decision tree system. We evaluated the resulting synthesized pruning algorithm on numerous synthetic data sets and found that it generates trees with seemingly better generalizing ability.

Keywords classification, automatic programming, meta-learning

Acknowledgements

First and foremost, I would like to thank my thesis supervisor Roland Olsson for the advice and support through the course of writing this thesis in addition to introducing me to such an interesting field as machine learning. I would also like to thank my fellow students for providing a great social and educational environment. Lastly, I thank family and friends for their support, especially my sister, Carina Hansen, for providing interesting suggestions and comments to this thesis.

Prerequisites

The reader should be familiar with functional programming, more specifically Standard ML, since ADATE, the automatic programming system utilized in this thesis, relies heavily upon Standard ML and a dialect of Standard ML called ADATE-ML. Still, it should be possible to follow most of this thesis except for some of the details about the inner workings of ADATE, the ADATE specifications and the programs synthesized.

However, no knowledge should be required about the other areas of machine learning and classification which this thesis touches upon since a description is provided of these areas. Nevertheless, the description is far from complete and the reader should consult the references given for more information. Thus, it is preferable that the reader has some knowledge of machine learning and classification.

Contents

Chapter 1

Introduction

Classification is one of the most popular and well studied fields of machine learning. The problem is how to predict which class an instance belongs to based on its attributes, for example predicting whether a mushroom is poisonous by considering its color and height. Systems making such predictions are known as classifiers, and are learned, that is induced, using already seen data where the class of each instance is known. In other words, classifiers make predictions about the future based on the past.

Classifier inducers search for the best possible classifier for a particular data set. Their search space is virtually infinite in terms of the number of classifiers, which means that it is practically impossible to conduct an exhaustive search that is guaranteed to find an optimal classifier. Therefore, the classifier inducers reduce the search space by favoring certain classifiers over others. This is known as the inductive bias of a classifier inducer.

It is the inductive bias and how well it fits a specific data set that determines the accuracy of the induced classifier. Thus, for a specific data set it is important to find the classification algorithm with the appropriate inductive bias.

Another paradigm in machine learning is automatic programming where a system automatically creates programs to solve a specific problem. The problem and some method for evaluating the generated programs are described in some sort of specification given to the system. This is declarative programming, distinct from normal programming in that the programmer specifies only what the problem is and not how to solve it. In this thesis, automatic programming is explored using Automatic Design of Algorithms Through Evolution (ADATE), which is one of the leading systems for automatic induction of programs.

In terms of ADATE and classification, there are two questions that are particularly interesting. First, is the inductive bias of ADATE better suited

in general than the inductive bias of other classifier inducers specialized for this task? ADATE is a more general system than traditional classification algorithms and it is more flexible in the type of classifiers it can produce. For example, in theory, it can produce classifiers such as decision trees, rule sets and neural networks.

Second, is it possible to adapt the inductive bias of classification algorithms to a specific domain or a specific type of domain using ADATE? This is different from inducing classifiers and is called meta-learning where the focus is on teaching the learner how to learn in the best possible way.

These two questions will be further explored in this thesis by letting ADATE create classifiers for specific data sets and letting ADATE improve an existing classification algorithm in the form of decision tree learning. In this way, the focus of this thesis is whether classification problems can be solved through automatic programming either directly or indirectly.

This thesis has the following outline. Chapter 2 introduces the sampling methods and the classification algorithms used in this thesis. Chapter 3 describes the ADATE system from how specifications are written to how programs are synthesized. Chapter 4 explains how ADATE can be used to synthesize classifiers for real world data sets, and the synthesized classifiers are compared with classifiers produced by other classification algorithms. Chapter 5 and chapter 6 describe how ADATE can be used to improve decision tree learning, and the resulting synthesized algorithms are evaluated empirically on both synthetic and real world data sets. We present in chapter 7 the most interesting areas of classification that should be further investigated through automatic programming and concludes this thesis in chapter 8.

Chapter 2

Background

This chapter gives background information covering the different methods and algorithms used later in this thesis. Most of this chapter is devoted to classification algorithms and a short introduction is given to the algorithms that are later compared to ADATE. However, these descriptions are not complete and the specified references should be consulted for more information.

2.1 Sampling Methods

Optimally, the amount of data available in training should be as large as possible to ensure there are enough data to both learn accurate classifiers and select the one that is best suited for a particular domain. However, data are often scarce and sampling methods are needed to increase the amount of data by creating several data sets from a single original data set. There are two major sampling methods, bootstrapping and cross-validation.

2.1.1 Bootstrapping

Bootstrapping performs selection with replacement on the original data set until a certain number of instances have been selected. It repeats this procedure until enough data sets are created. Each of the generated data sets is divided into a training and test set, meaning an instance may be included one or more times in the test sets.

2.1.2 Cross-validation

Cross-validation creates n data sets by dividing the original data set into n folds. It iterates over the folds, and adopts the current fold as the test set

and the remaining folds as the training set. Thus, each instance is included exactly once in the test sets.

The most extreme case of cross-validation is leave-one-out cross-validation where the number of folds is equal to the number of instances in the original data set. This allows almost all the data to be used in learning the classifier, but it is computational intensive.

It is popular to chose 10 folds when performing cross-validation.

2.2 Classification Algorithms

A wide range of classification algorithms have been developed through time with different underlying models and different theories of how a classifier should be built. As explained in the introduction, these algorithms have different inductive biases that affect their performance on a data set, and consequently, it is important to find the inductive bias that best fits the data set. This can be done empirically by applying a set of different machine learning algorithms and selecting the algorithm that performs the best.

This is an important methodology in machine learning that we used in [1](attached in appendix A for convenience) to find the most accurate algorithm for classifying infrasound signals. Initially, we trained only a single type of classifier, artificial neural network, using the Matlab neural network toolbox since we were most familiar with this type of classifier. However, we decided to execute a set of other machine learning algorithms from the Waikato Environment for Knowledge Analysis (WEKA) tool box as well to see how these performed on the same problem. Several of these algorithms were found to perform much better than the neural network algorithms used initially, which only shows the importance of evaluating many algorithms.

We decided to use WEKA in this paper as well to ensure accurate assessment of the programs synthesized by ADATE. WEKA is an open source data mining toolbox developed at the University of Waikato in New Zealand. It has support for all the tasks usually performed in data mining through numerous of algorithms for pre-processing, classification, regression and clustering. Since WEKA is written in Java, it will in theory run on any operating system with a recent version Java installed.

In the following sections, the WEKA algorithms employed in this paper are described. These are grouped according to how and what type of models they learn.

Note that some of the algorithms are classification algorithms only able to solve binary class problems or regression algorithms predicting numerical classes instead of nominal. Because these algorithms cannot solve classifica-

tion problems with multiple classes by themselves, they are executed using the ensemble algorithms MultiClassClassifier and ClassificationViaRegression, which will be explained in section 2.2.7.

2.2.1 Bayes

Bayesian algorithms are based on Bayes' Theorem, which is defined as

$$P(h|d) = \frac{P(d|h)Pr(h)}{P(d)}$$

where the h corresponds to a hypothesis, namely a prediction of a particular class, and the d represents the attributes of the unlabeled instance.

Naive Bayes

Naive Bayes (NB) assumes naively that every attribute, a_i, is independent of each other given the class. Thus, the likelihood of a class c is easily computed as follows:

$$P(c)\prod_i P(a_i|c)$$

The class with the highest likelihood is predicted. Although the independence assumption is seldom true for real world data sets, it seems to work reasonable well regardless.

2.2.2 Lazy

Lazy learning algorithms differ from the other classification algorithms in that the training of the classifier is postponed until classification. This allows the classifier to be customized according to each unlabeled instance at the expense of being computational intensive if there are many instances to classify.

IB1

IB1 [2] is a nearest neighbour algorithm that determines the class of an unlabeled instance according to the class of the nearest training instance. The distance between two instances are calculated using the euclidean distance

$$\sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

where a_i and b_i are the attributes i of the instances a and b.

IBk

IBk [2] is similar to IB1, but it uses the k nearest neighbours instead of only one. The predicted class is determined by the majority vote where each instance places a vote on its corresponding class. In this paper, we used leave-one-out cross-validation to determine the number of neighbours.

K*

K* [3] is a nearest neighbour algorithm that employs, instead the euclidean distance, an entropic distance function computing the probability of randomly transforming one instance into another. Each class receives a vote from each instance with a weight equal to the distance from it to the unlabeled instance, and the class with the most votes is selected.

Locally Weighted Learning

Locally weighted learning (LWL) [4, 5] selects a subset of the training instances, where each instance is weighted according to the unlabeled instance. A k nearest neighbour algorithm is applied to select the subset of instances, and the weight is calculated by a weighting function taking the euclidean distance as input. We chose to use the default linear weighting function $f(d) = 1 - d$.

The weighted training set is fed to another classification algorithm producing the final classifier. We chose to follow the recommendation in [6] and selected Naive Bayes.

2.2.3 Functions

These algorithms have mathematical or statistical foundations and create models that can be represented mathematically through functions. They are a mix of regression and classification algorithms.

Linear Regression

Linear regression (LinReg) is a standard linear regression algorithm that expresses the numerical class as a linear combination of the attributes. The coefficients of these attributes are calculated using the least-square method.

Logistic

Logistic builds logistic regression models and is implemented according to [7] with some modifications. These models have similar properties to linear

regression models, but the target attribute is transformed using the logit function and the weights are found by maximizing the log-likelihood instead of minimizing the sum of squared errors.

Simple Logistic

Simple Logistic (SLogistic) [8] also builds logistic regression models, but it uses another strategy than Logistic involving LogitBoost [9] and a base learner constructing simple regression models containing only the attribute yielding the minimum squared error. The number of boosting iterations used is determined by cross-validation.

MultilayerPerceptron

MultilayerPerceptron (MP) is a neural network algorithm that optimizes the weights of neural network using backpropagation. We used the default parameters for this algorithm yielding a forward feed neural network with three layers, input, hidden and output. The input layer comprises a bias node in addition to a node for each attribute after the nominal attributes have been converted to binary attributes. The hidden layer also contains a bias node in addition to n nodes determined by the following expression

$$\frac{i+o}{2}$$

where i and o is the number of nodes in the input and output layer. The output layer is composed of a node for each class.

The activation function for the nodes in the hidden and output layer is the sigmoid function.

RBF Network

RBF Network (RBFN) trains a radial basis function network, which is a type neural network. The network has three layers: an input layer with a node for each attribute; a hidden layer where each node has a Gaussian radial basis function as activation function, created using a clustering method called K-Means [10]; and an output layer containing a node for each class with sigmoid as activation function.

SMO

SMO, proposed by John Platt [11, 12], is a sequential minimum optimization algorithm for training support vector machines (SVM). The algorithm finds

the maximum margin hyperplane represented as a set of vectors known as support vectors. In order to solve non-linear problems with this linear classifier, the instance space is transformed using a non-linear kernel function. We chose to use the default polynomial kernel.

Voted Perceptron

Voted perceptron (VP) [13] transforms the input space using a polynomial kernel as SMO, but it uses the perceptron [14] algorithm to train the classifier. During training, it stores all the intermediate prediction vectors, namely the coefficients of the attributes, along with a weight of how many iterations they persisted without change. When classifying, each prediction vector votes on a class according to its weight, and the majority vote determines the predicted class.

2.2.4 Trees

These algorithms induce decision trees as classifiers, which basically contains two types of nodes: decision nodes and leaf nodes. Decision nodes are internal nodes containing a test on a specific attribute that determines which of the underlying branches an unlabeled instance should follow. Traversal continues from the root until a leaf node is encountered, and the leaf node predicts the class of the instance by utilizing a prediction function. Figure 2.1 shows a traditional decision tree where the leaves contain the class to predict.

Decision tree learning is composed of building and pruning. A decision tree is typically built by recursively selecting the most promising attribute and splitting the training set accordingly until all instances belong to the same class or all attributes have already been used. The most promising attribute is determined by the attribute maximizing the splitting criterion. The role of pruning is to simplify the decision tree either during or after building.

ID3

ID3 [15] is one of the first decision tree learners proposed, and it employs information gain as splitting criterion. Since it does not support continuous or missing attribute values, it can only solve a limited set of problems.

J4.8

J4.8 is an implementation of Quinlan's popular C4.5 [16] decision tree learner and it improves upon ID3 in several areas. First, it replaces the information

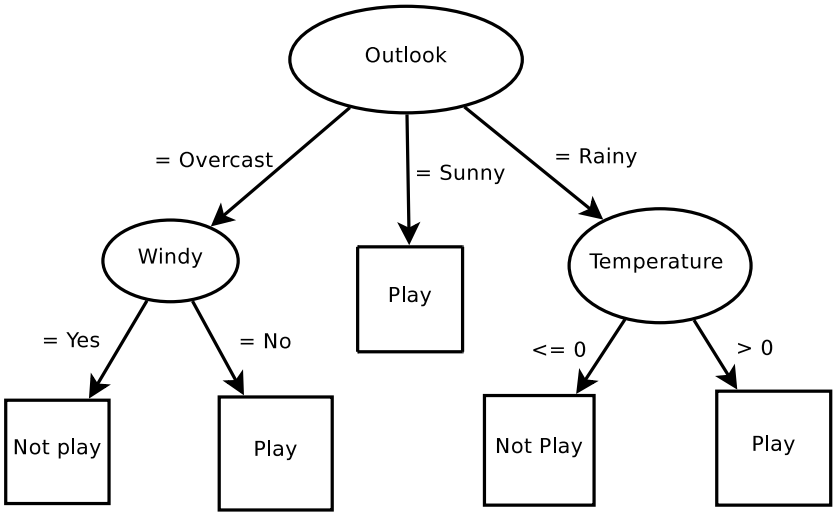

Figure 2.1: An example of a traditional decision tree where the ellipses are decision nodes and the rectangles are leaves.

gain splitting criterion with gain ratio since information gain favors attributes with many values. Second, it supports both continuous and missing values, and it performs pruning using error based pruning (EBP), which is described later in section 5.1.1.

REPTree

REPTree is a fast decision tree learner that resembles C4.5 in the implementation, but it uses information gain instead of gain ratio and reduce error pruning, described later in section 5.1.1, instead of EBP.

NBTree

NBTree [17] is a hybrid algorithm that creates decision trees with Naive Bayes classifiers at the leaves learned from the training instances reaching the node. It follows the standard decision tree learning algorithm and uses the mean accuracy of creating a Naive Bayes classifier at a given node according to 5-fold cross-validation as splitting criterion.

Logistic Model Trees

Logistic Model Trees (LMT) [8] builds decision trees with logistic regression models at the leaves, which are iteratively created using Simple Logistic explained in section 2.2.3. The trees are built similarly to C4.5 by selecting

attributes according to the gain ratio splitting criterion until there are no more attributes, all the instances have the same class or there are less than 16 instances. Pruning is performed using the pruning algorithm employed by the decision tree learner, CART [18].

M5'

M5' [19] is a reconstruction of Quinlan's M5 [20] that creates decision trees with linear regression models at the leaves. It chooses the attribute at each decision node that maximizes the standard deviation reduction of the class of the training instances reaching the node. When the tree is built, it traverses upwards from the leaves, while adding linear regression models at the nodes and possibly removing nodes if necessary. The predicted class value of an unlabeled instance is determined based on the output of all the linear regression models encountered when traversing the tree.

Decision Stump

Decision Stump (DS) induces simple decision trees, known as decision stumps, with only a single decision node. This node has a boolean test, which for a nominal attribute tests whether the attribute is equal to a specific value and for a continuous attribute tests whether the attribute is less or equal to a threshold. This algorithm is normally executed through ensemble algorithms like bagging and boosting.

Random Forest

Random Forest (RF) [21] uses bagging in combination with a random tree inducer. The random tree inducer builds a tree by choosing at a given node the best attribute among a set of randomly selected attributes. Bagging is explained later in section 2.2.7.

ADTree

ADTree [22] creates what is known as an alternative decision tree by using boosting to add the different branches. An alternative decision tree is, as illustrated by figure 2.2, simply a set of interconnected decision stumps with numerical leaves, where each leaf may be connected to a set of other stumps. The tree is used to classify unlabeled instances with binary classes by summing all the numerical nodes encountered while following the different paths of the tree applicable for the instances. The sign of this value determines the predicted class.

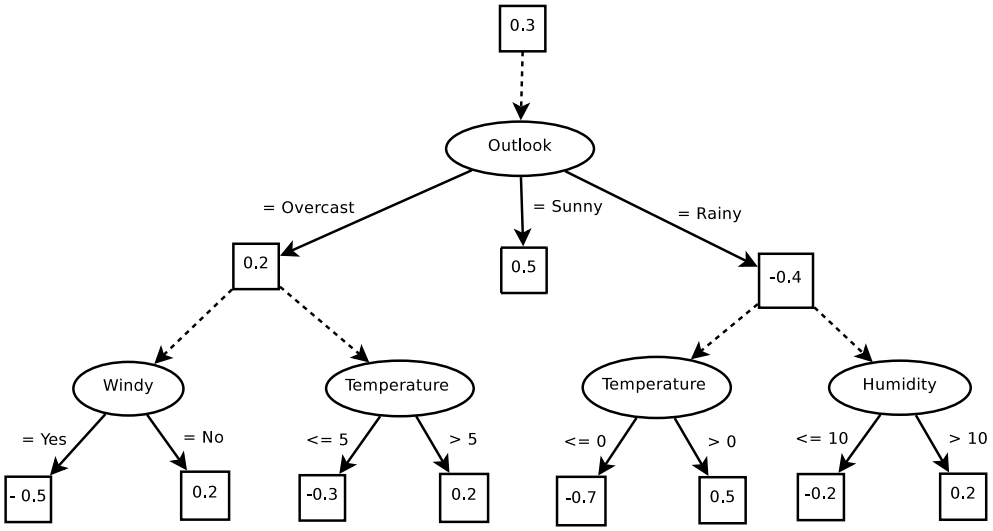

Figure 2.2: An example of an alternative decision tree where ellipses are decision nodes, rectangles are numerical leaves and dashed arrows connect the decision stumps together.

2.2.5 Rules

This group contains algorithms that create classifiers which are rule sets. Rule sets are intuitive and easier for humans to interpret than other classifiers like decision trees.

JRip

JRIP is an implementation of RIPPER [23] with some minor modifications added to fix what appear to be two bugs in the original algorithm [24]. It induces each rule of the final rule set in two steps. Firstly, the rule is grown by continually adding antecedents until it matches only training instances with a specific class. Secondly, the rule is iteratively pruned by processing the antecedents in reverse order.

OneR

OneR is a simple algorithm that creates a rule set for each attribute and chooses the rule set with the lowest error rate on the training data. Each rule set comprises a rule for each value of a particular attribute that predicts the majority class of the training instances matching the rule.

ZeroR

ZeroR is the simplest of all classification algorithms, and it only predicts the majority class of the training set. This algorithm provides an upper bound of the error rate that all other classification algorithms should be smaller than.

DecisionTable

DecisionTable (DT) [25] constructs a decision table classifier, which simply is a table containing the training instances with only a subset of their attributes included. The optimal subset of the attributes is found using best-first search combined with cross-validation where the DecisionTable algorithm is executed for different subsets. An unlabeled instance is classified as the majority class of the matching instances in the table, but if there are no matching instances, the majority class of all training instances is predicted instead.

PART

PART [26] creates a rule set by repeatably creating pruned decision trees using J4.8, converting them to rules and removing the training instances matching the rule until all training instances are covered by at least one rule. Each rule is created according to the path from the root of the decision tree to leaf covering the most training instances. In order to preserve computational resources, only partial decision trees are constructed where branches are expanded as needed.

M5Rules

M5Rules [27] builds regression rules using the same algorithm as Part except it generates trees using M5' instead of J4.8.

Ridor

Ridor is a RIpple DOwn Rule learner that first creates a default rule predicting the majority class of the training instances and then recursively adds exceptions to this rule until all training instances are classified correctly according to the rule set. A separate validation set is utilized to find the most accurate exception at each step.

NNge

NNge is a nearest neighbour algorithm forming non-nested general exemplars. A general exemplar is a hyper-rectangle that encompasses a set of training instances sharing the same class. In this way, each general exemplar is like a rule, and the nearest exemplar determines the class of an unlabeled instance. For more information please refer to [28, 29].

2.2.6 Misc

This group contains the algorithms that do not fit naturally into any of the other groups.

HyperPipes

HyperPipes (HP) is a simple and extremely fast classification algorithm that constructs a set of attribute ranges for each class. For nominal attributes, the range is the set of values observed for a particular attribute of the training instances matching a specific class. The range is found similarly for continuous attributes except the range is not a subset, but an interval ranging from the minimum to the maximum observed attribute value. Classification is performed by selecting the class with the most matching attribute ranges.

VFI

VFI [30] constructs a set of intervals for each attribute similarly to HyperPipes, but these intervals are not bound to a specific class. Thus, each interval contains a class count for each class according to the training instances that fall into it. Continuous attributes are basically discretised into a set of intervals, and an interval for nominal attributes is defined as a single attribute value. An unlabeled instance is classified using the majority vote, where each matching attribute interval is allowed to vote.

2.2.7 Ensemble

Ensemble algorithms use a base learning algorithm to create an ensemble of classifiers and combine these classifiers to reach a prediction. These algorithms differ in how the base learning algorithm is applied and how they combine the classifiers. The first two algorithms enhance the abilities of the base learner, making it possible to solve previously unsupported problems, while the last two enhance the performance of the base learning algorithm.

ClassificationViaRegression

ClassificationViaRegression allows a regression algorithm to solve classification problems. It creates a data set for each class using a 1-against-all encoding where the class is 1 if it is equal to the current class and 0 otherwise. A regression model is created for each class based on these data sets, and classification is performed by predicting the class belonging to the model yielding the greatest value.

MultiClassClassifier

MultiClassClassifier makes it possible to solve multi-class problems with algorithms that only support binary classes. This is possible through several methods, but we chose the default 1-against-all method explained in the previous section.

Bagging

Bagging [31] creates an ensemble of classifiers in order to increase the accuracy by stabilizing the base learning algorithm, or in other words decrease its variance. This is done by generating a set of "new" training sets using bootstrapping and applying the base learning algorithm on these data sets. Prediction is determined by the majority vote of the ensemble.

The success of bagging depends heavily on the properties of the base learning algorithm. It should be unstable, meaning that it is sensitive to small changes in the training set, so that its variance can be decreased.

In this paper, we let bagging create 10 classifiers using J4.8 as base learner.

AdaBoost.M1

AdaBoost.M1 [32] is a boosting algorithm that builds an ensemble of classifiers by forcing the base learning algorithm to focus on the instances that the previous classifiers had problems classifying correctly. This is done by accompanying every training instance with a weight representing the severity of misclassification such that the error rate is calculated as the sum of the weights of the misclassified instances divided by the sum of all the weights. Initially, each instance has equal weights, but after a new classifier is induced, the weights are updated so that misclassified instances increase in weight while the other instances decrease.

Majority voting is utilized to determine the class to predict and each

classifier votes with the following weight

$$-\log(\frac{e}{1-e})$$

where e is the error rate observed in training. Therefore, a classifier with a high error rate will not have as much impact on the final result as a classifier with a lower error rate.

In this paper, we use J4.8 as base learner and 10 boosting iterations, resulting in a ensemble of 10 decision trees.

Chapter 3

ADATE

ADATE [33, 34, 35] is a system for automatic programming researched and developed by Roland Olsson at Østfold University College. The system requires a specification describing what type of programs to create and how to evaluate them. According to this information, ADATE is able to automatically synthesize programs, which are written in a special language called ADATE-ML.

ADATE is considered a part of Evolutionary Computation (EC) and utilizes a population-based search inspired by natural evolution. The population, or the kingdom as it is called in ADATE, contains the most promising individuals, that is synthesized programs, at any given time. New individuals are created by selecting an individual from the kingdom and applying so-called compound transformations, a process called expansion. If some of these individuals seem promising, they will be inserted into the kingdom and possibly replace existing members.

The search is mostly systematic and thus differs from other methods in EC like genetic algorithms (GA) [36] and genetic programming (GP)[37]. Simple individuals are expanded before complex individuals and simple transformations are applied before complex transformations. This expansion process follows iterative-deepening [38]. Despite being systematic, the search is not exhaustive and heuristics are applied to reduce the search space of individuals to a maintainable size.

In this chapter, an overview is given of the different parts of ADATE. Firstly, a description is provided of the external parts of ADATE immediately exposed to the user, namely ADATE-ML and the specification. Secondly, the internal workings of ADATE is detailed by explaining the population management, the transformations and the overall search algorithm. Lastly, a relatively new feature in ADATE is introduced called habitats.

Table 3.1: Standard ML constructs illegal in ADATE-ML and how they can be represented in ADATE-ML. This table is taken from [39].

Construct	Standard ML	ADATE-ML

if expression	`if` E `then`
	  RHS1
	`else`
	  RHS2
		`case` E `of`
		  true => RHS1
		  | false => RHS2

Selectors	tl Xs	`case` Xs `of`
		  nil => `raise` NA1
		  | cons(X1, Xs1) => Xs1

Boolean operator	E1 `andalso` E2	`case` E1 `of`
		  true => E2
		  | false => false

Left Hand Side	`fun` len nil = 0
	| len(X1::Xs1) =
	  1 + len Xs1
		`fun` len Xs =
		  `case` Xs `of`
		  nil => 0
		  | cons(X1, Xs1) =>
		  1 + len Xs1

Variable declaration	`let`
	  `val` V = E1
	`in`
	  E2
	`end`
		`case` E1 `of`
		  V => E2

3.1 ADATE-ML

ADATE-ML is a purely functional language specifically designed for synthesis, and the programs synthesized by ADATE are written in this language. It is a subset of Standard ML, where numerous features have been removed to keep the language as compact and simple as possible to make synthesis easier and more effective.

Many of the features removed have little impact on the expressiveness of the language. This is a result of great redundancy in Standard ML so that one construct may be equivalently represented using other constructs. For instance, constructs such as if expressions, selectors, boolean operators, variable declarations and functions declared with alternative left hand syntax can be represented with case expressions, which is why these constructs are illegal in ADATE-ML. Table 3.1 shows these constructs and the alternative syntax in ADATE-ML.

Other features, on the other hand, are removed although there are no equivalent alternative representations. This includes features such as poly-

morphic data types, structures, signatures, functors and curried, anonymous and higher-order functions. These features would complicate the synthesis process if introduced, and the transformations applied to the programs would have to be redesigned.

Similarly to the core language, the size of the built-in library of data types and functions is small. The data types included are only `bool`, `real`, `integer`, `array` and `rconst`. Most of these are familiar from Standard ML except `rconst`, which is used by ADATE to define constants. The built-in functions consist of arithmetic and boolean functions for modifying and comparing `integer`s and `real`s as well as functions for accessing and modifying elements in an `array`.

3.2 Specification

An ADATE specification describes the problem that the ADATE system should solve, that is what type of program to synthesize. The specification consists of two parts separated by %%, where the first part is written in ADATE-ML and the second part is written in Standard ML. These two parts are explained in the next sections. The first two sections belong to the ADATE-ML part, while the last two sections belong to the Standard ML part.

3.2.1 User-Defined Data Types and Functions

Since ADATE-ML only contains a very limited set of data types and functions, there is almost always a need to define additional data types and functions to be able to represent the problem at hand. This is done in exactly the same manner as in Standard ML using the `datatype` and `fun` keywords.

However, defining a function does not automatically make it available during synthesis; Instead, this has to be specifically declared for each function in the Standard ML part, which is covered later in section 3.2.3.

3.2.2 The f Function

The `f` function is the function or the program that ADATE should synthesize. Typically, this function is initialized to an almost empty function, which raises only an exception and never returns anything, since little is known about the implementation of the final program. Nevertheless, it is possible to define a non-empty `f` function that already implements an existing algorithm

in order to allow ADATE to improve it instead of starting from scratch. Both approaches are utilized in the specifications written for this paper.

The `f` function is not called directly by ADATE. Instead, it calls another function, `main`, which is required to call the `f` function at least once. This allows the `f` function to be a standalone algorithm or a part of a larger algorithm. In this paper, the `f` function is always a standalone algorithm and the `main` function simply wraps the `f` function.

3.2.3 Available Functions

The functions available to ADATE during synthesis of the `f` function are controlled through two lists. The first list, `funs_to_use`, contains the names of the help functions and constructors ADATE is allowed to use. The second list, `abstract_types`, restricts access to the listed data types by preventing pattern matching against their constructors.

These two lists should be defined with great care since the functions available in synthesis directly affect the ability of ADATE to synthesize programs. If the available functions do not include all the necessary functions, it will make it impossible for ADATE to reach a good `f` function. If, on the other hand, there are too many functions available, it will increase the number of synthesis choices to the extent that it will be virtually impossible to reach a good `f` function within a reasonably amount of time.

Thus, the functions available should be few and at the same time contain all the necessary functions. This can be hard to achieve in practise since it often is difficult to decide which functions that will be relevant in synthesis. Luckily, ADATE is not that sensitive to the available help functions because it can invent auxiliary functions as needed.

3.2.4 Inputs and Output Evaluation Function

Evaluation of the synthesized programs is performed through a set of inputs and the output evaluation function, `output_eval_fun`, defined by the user. The inputs are iteratively given to the `main` function, which calls the `f` function one or more times before returning some kind of output. Each input and output pair is given to the output evaluation function, which calculates an evaluation value. These evaluation values are combined to produce a single evaluation value for the outputs.

Each evaluation value is a record containing three values. The first two values are integers and represent how many correct and wrong decisions made by the `f` function. The last value is a grade, used to introduce finer granularity to enable differentiation of programs that have the same number

of correct and wrong decisions. An empty grade can be returned if this finer granularity is unneeded.

Since ADATE relies upon a separate user-defined function to provide the evaluation value of the synthesized programs, the evaluation procedure can be customized specifically to the specification. For instance, the evaluation function can be defined to support the popular evaluation approach, where the output returned from the synthesized program is compared to what is believed to be the correct output of the input. However, it is not always apparent what the correct output is or there might be more than one correct output. In these circumstances, a more complex evaluation function is needed. Regardless of implementation, the evaluation function should be relatively effective because it is called for every synthesized program and input.

ADATE utilizes two sets of inputs, one for training and one for testing. The training inputs are used during synthesis to manage the population. Naturally, the fitness value for this data is overly optimistic and might be a result of overfitting, making it unsuitable for evaluating the generality of a program. Therefore, a separate test set is used instead for this purpose.

There are at least three usages of this test set. First, it enables discovery of overfitting in the population by comparing fitness values of the programs for the training and test set to see whether an increase in training results in an increase in testing as well. Second, it can be used to select the program in the population that is likely to perform the best in general. Third, it allows the synthesized programs to be compared with other systems producing similar programs.

3.3 Population Management

In this section, an overview is given of how the kingdom is maintained in ADATE by explaining how the individuals are evaluated, how the kingdom is structured, and how new individuals are inserted into the population.

3.3.1 Evaluation of Individuals

Each individual is evaluated on a set of inputs, namely the training set or the test set, in terms of syntactic complexity, time complexity and performance. The syntactic complexity, intuitively, represents the size of the individual, that is the number of bits needed to represent it. There are three different functions for calculating the syntactic complexity in ADATE [33].

$$pe_1 = -N_c :: G :: [N_w, S, T]$$
$$pe_2 = -N_c :: G :: [N_w, T, S]$$
$$pe_3 = [N_w, N_c] \ @ \ G \ @ \ [S, T]$$

Figure 3.1: The lists returned from three program evolution functions in ADATE.

The time complexity, naturally, relates to the time it takes to execute the individual for the data set. However, instead of measuring the actual time, which is unreliable, it counts the number of calls made to either the `f` function or the auxiliary functions defined in `f`.

The performance is computed using the two previously described complexity measures together with the concatenation of all the values returned from the output evaluation function. This amounts to the following five values: T, the time complexity; S, the syntactic complexity; N_C, the number of correct; N_W, the number of wrong; and G, the grade. These values are ordered differently and returned as a list by the three program evaluation functions defined in ADATE. Figure 3.1 presents the different orderings of these lists.

Two performance values are compared to each other lexicographically. As a result, the elements are compared successively to each other until two elements differ. Note, smaller is considered better.

3.3.2 Kingdom Structure

The kingdom comprises nine sub-populations, one for each combination of the three syntactic and three program evaluation functions. Each sub-population is divided into cells, as illustrated by figure 3.2, according to a time and syntactic complexity grid, where each cell contains what is known as a family.

A family consists of an individual called the base individual and two collections of individuals called embed genus and output genus. The base individual is the individual that currently has the smallest time and syntactic complexity for a given cell. The other two collections are directly related to the base individual. An embedded genus comprises individuals transformed from the base individual using the embedded transformation, which will be introduced in section 3.4.1. An output genus, on the other hand, includes individuals that are semantically different than the base individual. The diversity in an output genus is maintained using the shark pool algorithm

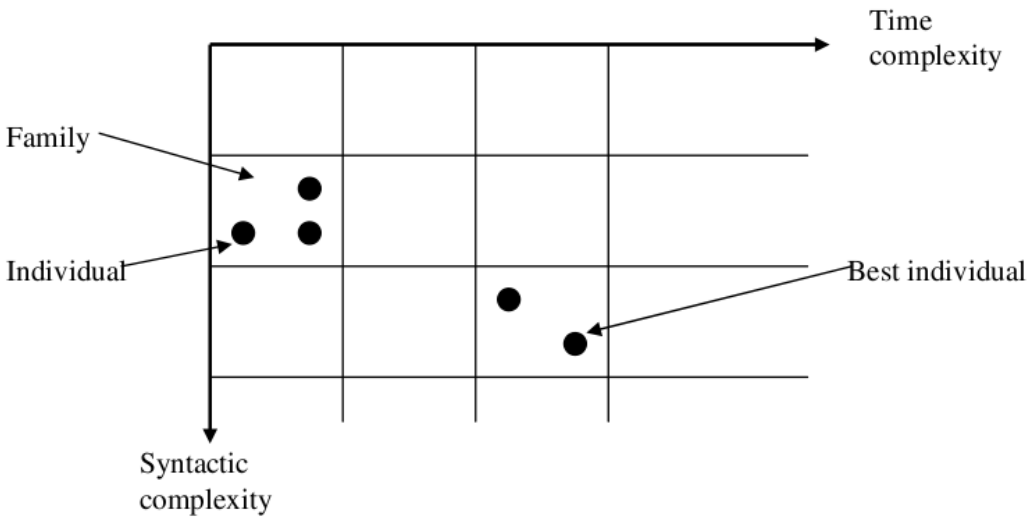

Figure 3.2: The grid that each sub-population is divided into. The figure is taken from [39]

described later in section 3.6.2.

The nine sub-populations are not orthogonal and individuals may appear in any of them at any given time. This typically results in a considerable overlap between the different sub-populations, meaning the cardinality of the kingdom is often much smaller than the sum of the cardinalities of the different sub-populations.

3.3.3 Insertion into the Kingdom

The main design goal of the insertion algorithm is to prevent missing links in the genealogical chains of the individuals in the population [35]. A missing link is defined as a gap in the genealogical chain that is too large for one ancestor to reach the next ancestor through a set of transformations reasonably limited in complexity. Missing links should be avoided since they might result in ADATE getting stuck in local minima.

Based on hand-constructed genealogical chains of individuals and empirical studies, a heuristic was developed that seems to avoid missing links reasonably well in a population. This heuristic states that all individuals in the kingdom should be better than all individuals in the population that are not bigger. In addition to preventing missing links, it keeps the population small and limits the introduction of individuals containing no meaningful improvements.

This heuristic forms the basis of the algorithm used to insert individuals

into the different sub-populations of the kingdom. An individual is inserted into its corresponding family and replaces the base individual if the following two assertions hold: it is smaller than the base individual and it is better than all the base individuals without a larger time or syntactic complexity. In addition to replacing the base individual of the family, it replaces any base individual that performs worse and has not a smaller time or syntactic complexity.

If, on the other hand, the individual could not replace the base individual of the family, an attempt is made to insert it into an embedded or output genus. These collections are included to increase the diversity in the kingdom and decrease the possibility of missing links.

3.4 Transformations

ADATE synthesizes new programs based on existing programs through compound transformations, where a single compound transformation comprises a series of atomic transformations. Most of these atomic transformations are specifically designed with program synthesis in mind and do not directly emulate concepts in the nature such as mutation and cross-over. Both types of transformations are described next.

3.4.1 Atomic Transformations

Atomic transformations are the smallest transformation unit in ADATE and there are currently six atomic transformations available.

Replacement

Replacement replaces an existing expression with a new synthesized expression.

Replacement without Degradation

Replacement without degradation is a special type of Replacement transformation that does not degrade the `f` function. This is determined according to a special performance measure $-N_c :: Grades@[N_w]$, similarly defined as the measures in section 3.3.1.

Case Distribution

Case Distribution transforms case expressions. If the case expression is located inside a function call, that is the result of the case expression is one or more of the parameters to the function, it will move the case expression outside the function call, so that the function is called for each clause in the case expression. This transformation can also be conducted in opposite direction by moving the case expression inside a function call.

Abstraction

Abstraction converts an expression into a let-function by moving a part of the expression inside the function definition and the rest of the expression inside the call to this function at the end of the let expression. This operation is basically the opposite procedure of inlining, where a function call is replaced by the actual expression in the function.

This transformation enables ADATE to create auxiliary functions by itself. It reduces the need for help functions and decreases the responsibility of the specifier in defining them. In addition, the number of synthesis options can be reduced because the auxiliary functions are locally scoped and fewer help functions are required.

Embedding

Embedding generalizes the signature of a function by either changing the input type or the return type.

Cross-over

Cross-over applies a genetic algorithm to a set of REQ transformations where each one is considered an allele.

3.4.2 Compound Transformations

As explained previously, a compound transformation consists of a series of atomic transformations. Only the first atomic transformation is chosen freely, while the other transformations are constrained by a set of predefined coupling rules.

There are two types of coupling rules in ADATE: weak and strong coupling rules. A weak coupling rule specifies a pre-condition that must be true for some of the previous transformations. A strong coupling rule, on the

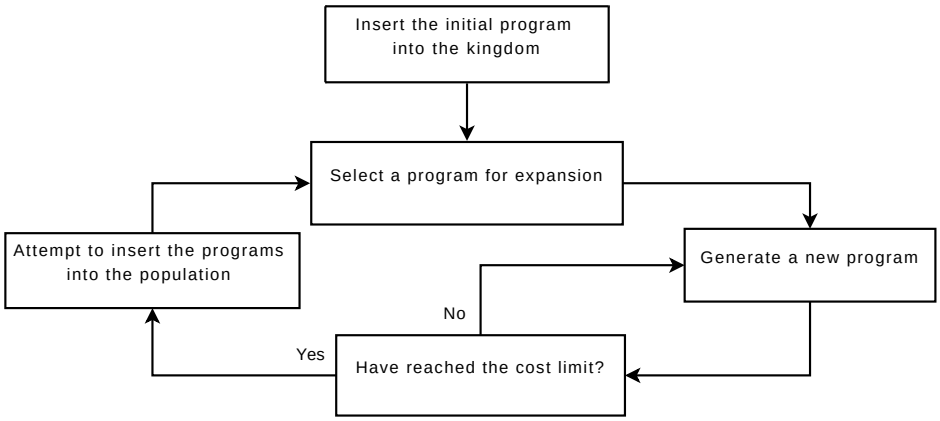

Figure 3.3: A graphical representation of the overall search algorithm in ADATE.

other hand, is more strict and the pre-condition must be true for the previous transformation. Since each rule can only be applied once, there is a finite set of compound transformations.

3.5 Overall Search

ADATE searches for new individuals to insert into the kingdom by expanding the individuals already in it through compound transformations. Initially, the kingdom contains only the start program, which naturally is the first program to be expanded. Expansion of the different programs is an infinite process, and it is the responsibility of the user to stop ADATE based on the state of the kingdom. Figure 3.3 presents a graphical overview of the search algorithm.

ADATE maintains a cost limit determining the number of individuals to synthesize for a specific individual. It is divided equally between the compound transformations, meaning the number of individuals synthesized for each transformation is the cost limit divided by the number of compound transformations. Initially, it is set to 1000.

In addition to the cost limit, ADATE maintains a collection of individuals eligible for expansion for the current cost limit. The simplest individual in this collection is selected for expansion and the resulting synthesized individuals are possibly inserted into the kingdom depending on their properties. After expansion, the individual is removed from the collection.

When the collection is empty, the cost limit is multiplied by 3.6 and the individuals expanded for lower cost limits are allowed to be expanded again.

This is called iterative-deepening where the cost limit first is increased when all eligible expansions for a specific limit have been considered.

Iterative-deepening is also used during expansion for a specific compound transformation, where simple transformations are applied before more complex transformations. Because of iterative-deepening and how the kingdom is managed, it is clear that ADATE has an inductive bias that favors simple individuals over complex individuals.

3.6 Habitats

Habitats is a new feature in ADATE inspired by geographic speciation, also known as Allopatric speciation in biology, where populations are isolated from each other geographically and evolve separately. Thus, the same specie may take different evolution paths according to the specific habitat they live in. This has been observed in nature, and probably the most famous observation was made by Charles Darwin on the Galapagos Islands where he found different but related specifies of finches with different beaks specialized to the food available in their habitat.

ADATE emulates geographic speciation by creating and managing five separate habitats. Each habitat has a separate kingdom and different training and validation sets. The data sets are created by applying 5-fold cross-validation on the original data provided to ADATE, meaning there is no overlap between the validation sets. Hopefully, this difference in data allows evolution to take different paths for the different habitats.

Geographic separation of habitats in nature are not eternal and may break down for some reason, allowing individuals from one habitat to migrate to other habitats. This same notion of migration is adopted in ADATE, where migration across habitats is permitted at different intervals. In that way, the habitats can evolve separately as well as share discoveries with each other.

The success of habitats in ADATE depends on how well the individuals perform in general. Accordingly, selecting an individual that performs well in only one habitat is not preferable. Instead, some method must be applied to either find an individual or a combination of individuals that are likely to perform well in general. There are currently four such methods included in ADATE, where one of them, ADATE boost, has been developed for this thesis.

3.6.1 Committee

Committee selects the individual with the best performance on the validation set for each habitat, resulting in a committee with as many members as there are habitats.

Voting is used to determine the output of a committee. Each individual votes on its output and the output with the most votes is selected.

3.6.2 Shark Pool

Shark pool is similar to Committee in that both methods create a committee of individuals and that they both use the same voting procedure to determine the output of the committee. However, shark pool does not select one individual from each habitat. Instead, it extracts the individuals considered relevant from all the habitats and places them in turn into a limited sized pool.

If the limit of the pool is exceeded after insertion, one of the individuals in it is discarded. This is determined by feeding them food in the form of inputs and removing the individual that eats the least amount of food. The size of the pool is equal to the number of members in the committee, and the final committee consists of the individuals remaining in the pool after inserting all the relevant individuals

All inputs contain the same amount of food, but an input is equally divided between the individuals that eat it. Eating is allowed if the individual returns the correct output for the input. In this way, the diversity of the pool is maintained by favoring individuals that perform well on different inputs.

Shark pool uses three different methods, C, SR, and BC, for extracting the relevant individuals from the different habitats. C extracts all individuals from the habitats, but removes duplicate individuals. SR is identical to C except individuals considered too big are excluded. BC, on the other hand, extracts only the base individuals from the habitats while removing duplicates and enforcing the same size restriction as SR. Of these three, BC appears to create the most accurate committees.

The size of the shark pool is varied between 2 and 20.

3.6.3 ADATE Boost

ADATE boost is based on ADABoost.M1 and constructs a committee of individuals from the different habitats using a weighted data set. However, instead of training each member of the committee specifically to the current weights, it only selects the individual from the habitats that is the

best according the weights. This fits better into the systematic and infinite population-based search utilized by ADATE. The source code of ADATE boost is available in appendix B.

The weights of the inputs are maintained as a distribution and always sum to one, meaning the error rate of an individual is simply the sum of the weights of the wrongly predicted inputs. Initially, the weights have the same value since every input is regarded equally hard. However, they are updated for each individual selected to be a member of the committee by first multiplying every wrongly predicted input with $\frac{e}{1-e}$ where e is the error rate, and then normalizing all the weights so the sum of all weights is one. Consequently, inputs correctly predicted decrease in weight, while the wrongly predicted inputs increase.

If the currently selected individual has an error rate greater than 0.5, it is discarded and any further construction of the committee is terminated. This is done in order to guarantee that the training error of the committee approaches zero as the number of individuals increases [40]. This guarantee of the training error is meaningless if it has no relevance to the error in general. However, it can be proven that the training error and the general error will be close as long as the committee is not too "large" and the individuals are not too "complex" [40].

The output of a committee is determined identically to ADABoost.M1 using majority voting, where each individual votes with a weight calculated according to their error rate e as follows: $-\log(\frac{e}{1-e})$.

The maximum number of members in the committee is varied between 2 and 20.

3.6.4 Joint Selection

Joint selection differs from the other three methods discussed in that it instead of forming a committee only selects a single individual. Selection is performed in two steps. First, a filtering procedure is applied to remove all the individuals that do not appear in at least a certain number of habitats. This filtering procedure is based on the belief that an individual which appears and survives in different habitats has a greater chance of survival in general than an individual appearing in only one. Second, the individual with the best performance among the remaining individuals is selected.

The number of habitats the individuals must appear in is varied between two and five.

3.7 Summary

In this chapter, an introduction to the ADATE system has been given. Both the external parts, ADATE-ML and the specification, and the internal parts, population, transformations, the overall search and the habitats, have been described, so that it should be possible to follow and understand the specifications and synthesized programs presented later in this thesis as well as having an understanding of how ADATE synthesized these programs.

Note, several versions of ADATE have been utilized in this thesis, and the habitats explained in the last section are only applicable to the version of ADATE utilized in the next chapter where ADATE is applied directly to classification.

Chapter 4

Synthesizing Classifiers

In this chapter, ADATE is applied directly to a set of classification problems in order to synthesize classifiers. The specifications are generated automatically and the traditional attribute-value encoding is used. In this way, the usage of ADATE is similar to the usage of other classification algorithms, and it is from this perspective it is compared to both simple and state-of-the-art classification algorithms.

This chapter is outlined as follows: First, a description is given of the data sets that ADATE is applied to. Second, the specification generation procedure is explained in addition to how the specifications were executed and the final classifiers were selected. Lastly, ADATE is compared to other classification algorithms, first empirically according to accuracy and then according to a set of other properties.

4.1 Data Sets

The 10 data sets are taken from the UCI machine learning repository [41]. They have a varying number of classes and contain a mix of continuous and nominal attributes. Table 4.1 shows the properties of these data sets.

Thyroid Disease (all)

This data set was contributed by the Garavan Institute and J. Ross Quinlan. The problem is to diagnose a patient for thyroid disease based on age, sex and set of different health measures. There are 3772 instances, 4 classes, 6 continuous attributes and 22 nominal attributes. An additional attribute is ignored since all its values are missing.

Table 4.1: Properties of 10 real world data sets.

Name	# Instances	# Attributes	# Continuous	# Nominal	# Classes
all	3772	28	6	22	4
cmc	1473	9	2	7	3
dna	3186	60	0	60	3
ger	1000	20	7	13	2
pag	5473	10	10	0	5
sat	6435	36	36	0	6
seg	2310	18	18	0	7
spa	4601	57	57	0	2
veh	846	18	18	0	4
yea	1498	8	7	1	10

Contraceptive Method Choice (cmc)

The data donated by Tjen-Sien Lim are based on the 1987 National Indonesia Contraceptive Prevalence survey. In this survey, married women, either not pregnant or not aware of it at the time, were interviewed about which of the following three types contraception they used: no, short term or long term contraception. The problem is to predict which of these contraception types a woman would prefer based on demographic and socio-economic data. There are 2 continuous attributes, 7 nominal attributes and 1473 instances.

Splice-junction Gene Sequences Database (dna)

G. Towell, M. Noordewier, and J. Shavlik donated this data base where the instances consist of DNA sequences of 60 base-pairs, and the aim is to determine what type of boundary there is in the middle of these sequences with regard to splice-junction. Splice-junction involves the process of splicing proteins together, where one part is removed (intron) and one part remains (exon). As a result, there are three different boundary types: intron-exon, exon-intron and none. The total number of instances is 3186.

German Credit Data (ger)

This data were contributed by Professor Dr. Hans Hofmann. The objective is to assess an application for credit as either good or bad based on demographic and economic data supplied in the form of 13 continuous and 7 numerical attributes. All in all, there are 1000 instances.

Page-blocks (pag)

This data set was created by Donato Malerba using a segmentation procedure to extract the different blocks that compose a document. The aim is to classify a block as either text, horizontal line, picture, vertical line or graphics. Each block is described by 10 numerical attributes, and the data set consists of 5473 blocks extracted from 54 different documents.

Landsat Satellite (sat)

This data set provided by Ashwin Srinivasan was generated using only a small portion of the original Landsat Multi-Spectral Scanner imagery purchased from NASA. Each instance is a 3x3 area of pixels in the imagery where the aim is to determine the type of scenery the central pixel represents. Since the satellite imagery contains four different spectral bands(two visible, and two (near) infrared), each pixel is represented by four different values ranging from 0 to 255, making the total number of attributes 36. Moreover, there are 6 different classes and 6435 instances.

Image Segmentation (seg)

The Vision Group at University of Massachusetts created the data set by randomly selecting instances from seven outdoor images. These images were segmented pixel by pixel into one of the following classes: brickface, sky, foliage, cement, window, path or grass. Similarly to Landsat Satellite, each instance is a region of 3x3 pixels, but the instances do not contain the actual pixel values as attributes. Instead, the instances contain different color means together with other attributes describing the region. Furthermore, there are 18 continuous attributes and 2310 instances.

Spambase (spa)

The data contain 4601 spam and non-spam e-mails collected by Mark Hopkins, Erik Reeber, George Forman, Jaap Suermondt at Hewlett-Packard Labs, 1501 Page Mill Rd., Palo Alto, CA 94304. Not surprisingly, the task is to determine whether an e-mail is spam or not. Each e-mail is described by 57 continuous attributes, where 48 of these hold the relative frequency of specific words, 6 include the relative frequency of specific characters, and the last 3 attributes contain the average length, the max length and the total length of consecutive capital letters.

Vehicle Silhouettes (veh)

The data set is based on a collection of silhouettes gathered at Turing Institute, Glasgow, Scotland by taking 846 pictures of different vehicles at varying angles and orientations. These pictures were processed and 18 continuous attributes describing each shape were extracted.

The object is to predict what kind of vehicle these attributes represent. There are four different vehicles in the data set, a double decker bus, a Chevrolet van, a Saab 9000 and an Opel Manta 400, where the SAAB 9000 and the Opel Manta 400 are the most similar in shape and should be the most difficult to distinguish from each other.

Yeast (yea)

This database was donated by Paul Horton, and the aim is to determine the localization of a yeast protein within a cell. There are 10 classes, 7 continuous attributes, 1 nominal attribute and 1498 instances.

4.2 Specifications

A separate specification is generated for each of the 10 data sets using the application, C5conv, which is bundled with ADATE. C5conv is used similarly to the popular C4.5 decision tree system and operates on the same files, namely a names file describing the type of the attributes and a data file containing the instances. Based on these two files, C5conv generates an ADATE specification for the data set automatically without needing any knowledge of ADATE-ML or the inner workings of ADATE.

In the following sections, a general description is provided of how C5conv defines the different parts of an ADATE specification. Some of the parts are exemplified with excerpts from the specification generated for the thyroid disease data set in order to make the description easier to follow. This specification is presented completely in appendix C.

4.2.1 User-Defined Data Types

C5conv defines data types based on the attributes defined in the C4.5 names file. Nominal attributes are defined as distinct data types and have a constructor for each nominal value. Continuous attributes, on the other hand, are represented using the built-in `real` data type. If any of the attributes contain missing values, a separate constructor is defined for the missing values. Figure 4.1 presents an example of three data types, where the first is a

```
datatype age = age_NONE | age_SOME of real
datatype sex = sexM | sexF | sex_NONE
datatype on_thyroxine = on_thyroxinef | on_thyroxinet
```

Figure 4.1: Three of the data types defined by C5conv for the thyroid disease data set.

```
(age_SOME ~0.3436123348017621, sexF, on_thyroxinet,
    query_on_thyroxinef, on_antithyroid_medicationf, sickt,
    pregnantf, thyroid_surgeryf, I131_treatmentf,
    query_hypothyroidf, query_hyperthyroidf, lithiumf,
    goitref, tumorf, hypopituitaryf, psychf, TSH_measuredt,
    TSH_SOME ~0.4968018566212889, T3_measuredf, T3_NONE,
    TT4_measuredt, TT4_SOME ~0.24065420560747663,
    T4U_measuredt, T4U_SOME ~0.21014492753623187,
    FTI_measuredt, FTI_SOME ~0.16921119592875317,
    TBG_measuredf, referral_sourceother )
```

Figure 4.2: One of the inputs for the thyroid disease data set.

continuous attribute with missing values, the second is a nominal attribute with missing values and the third is a nominal attribute without missing values.

In addition, a data type is defined for the class attribute in exactly the same manner as nominal attributes.

4.2.2 Inputs and Output Evaluation Function

The training and test inputs are created from the C4.5 data file by representing each instance as tuple and converting all attributes to their corresponding ADATE-ML data type. In addition, all continuous values are scaled to a value between -0.5 and 0.5 because of the way ADATE generates constants. Figure 4.2 shows an example of a single instance for the thyroid disease specification consisting of both continuous and nominal attributes.

Each input has a corresponding output in the form of a class. The output is used by the evaluation function to determine whether a program is able to classify an instance correctly, and the fitness value of a program is simply the number of correct and wrong classifications made.

```
fun f( (X0age, X1sex, X2on_thyroxine, X3query_on_thyroxine,
        X4on_antithyroid_medication, X5sick, X6pregnant,
        X7thyroid_surgery, X8I131_treatment,
        X9query_hypothyroid, X10query_hyperthyroid,
        X11lithium, X12goitre, X13tumor, X14hypopituitary,
        X15psych, X16TSH_measured, X17TSH, X18T3_measured,
        X19T3, X20TT4_measured, X21TT4, X22T4U_measured,
        X23T4U, X24FTI_measured, X25FTI, X26TBG_measured,
        X27referral_source ) :
        age * sex * on_thyroxine * query_on_thyroxine *
        on_antithyroid_medication * sick * pregnant *
        thyroid_surgery * I131_treatment *
        query_hypothyroid * query_hyperthyroid * lithium *
        goitre * tumor * hypopituitary * psych *
        TSH_measured * TSH * T3_measured * T3 *
        TT4_measured * TT4 * T4U_measured * T4U *
        FTI_measured * FTI * TBG_measured * referral_source
    ) : therapy =
        raise D1
```

Figure 4.3: The initial `f` function for the thyroid disease data set.

4.2.3 The f Function

The `f` function takes an instance as input, and returns the class of the instance given in. Initially, it is an almost empty function raising only an exception. Figure 4.3 presents the initial `f` function for the thyroid disease data set.

4.2.4 Available Functions

ADATE is permitted to use functions that operate on the built-in data types `rconst` and `real`. The `rconst` functions include `tor` and `rconstLess`, where `tor` converts a `rconst` to a `real` and `rconstLess` returns whether a `real` is less than a `rconst`. The `real` functions consist of `realLess`, which returns whether a `real` is less than another, and the typically arithmetic functions `realAdd`, `realSubtract`, `realMultiply` and `realDivision` in addition to the `sigmoid` function, which squashes a `real` value into to the interval between 0 and 1.

Furthermore, ADATE is allowed to use the constructors for the `bool` data type and data type for the class.

4.3 Execution of ADATE

The 10 ADATE specifications were executed on a cluster with 26 computers over a period of approximately three weeks. The 26 computers have a total of 40 CPU cores that are a mix of Intel Pentium D and Pentium 4, meaning each specification was executed on 4 cores.

These specifications were executed using a special version of the ADATE system utilizing habitats as described in section 3.6. This version employs four different algorithms to select either a single individual or a set of individuals from the different habitats.

However, which of these algorithms should be used to select the final classifier for a specific data set? One solution would be to select the classifier that is the most accurate on the training set. Unfortunately, the results from the training data are overly optimistic and a low error rate may be a result of overfitting. Another solution would be to employ a separate validation set, but this would decrease the number of instances, possibly yielding less accurate individuals.

Instead of using any of these solutions, we decided to choose the committee with 15 base individuals produced by the shark pool algorithm. This heuristic is based on observations made during the initial testing of the version of ADATE with habitats, where it seemed to select the most accurate committees.

4.4 Results and Analysis

ADATE is compared to 32 classification algorithms across the 10 data sets previously described. These algorithms have previously been introduced in section 2.2, and they are a part of the WEKA machine learning toolbox.

When executing these classification algorithms and ADATE, their default parameters were utilized and no specific tuning was conducted. By avoiding tuning, we hope to keep the execution as simple and fair as possible. However, this means the performance of the algorithms may be improved, especially for the more complex classification algorithms such as the neural network algorithm, MultilayerPerceptron.

The data sets utilized in the comparison are divided in half into a training and test set. Optimally, cross-validation should have been used to allow utilization of all instances in testing and to ensure that the observed accuracy is not a result of a preferable composition of the training set. However, the high computational requirements of the ADATE system made it impossible to execute ADATE several times for each data set within the time available

for this thesis.

ADATE and the other classification algorithms are compared over multiple data sets using the Friedman test [42, 43, 44], which will try to reject the null hypothesis of all algorithms producing equally accurate classifiers. It is rank based and the algorithms are ranked for each data set according to the accuracy of the classifier they produce, where the most accurate classifier receives the highest rank, rank 1. If two or more classifiers are equally accurate, the average rank is assigned to them.

If the null hypothesis is rejected at significance level 0.05, the post hoc test proposed by Holm [45] is employed to investigate which of the classification algorithms that are truly different from ADATE.

Table 4.2 contains the ranks given to each of the classification algorithms for each data set. The last column includes the average rank of each algorithm for all the data sets, where a + means the algorithm is significantly better than ADATE and a - means it is significantly worse. In addition, the error percent of each algorithm for the different data sets is shown in table 4.3.

ADATE and LMT have the highest average rank across the 10 data sets with a rank of 7.5. They perform relatively well for most of the data sets and have never a rank worse than 14. However, ADATE has only a rank higher than three in two of the data sets, and LMT has only a rank higher than three in three of the data sets. Furthermore, the difference in average rank is not that great between the algorithms, and ADATE is only significantly better than the eight worst algorithms.

Although ADATE and LMT have the highest average rank, it is interesting to note that they have far from the highest rank for each of the data sets. Instead, it varies considerably from data set to data set which of the algorithms that performs the best. For instance, K*, J4.8 and LWL have in turn the highest rank for one data set, despite having low ranks for the other nine data sets. These observations confirm the no-free-lunch theorem [46], which states there is no universal classification algorithm producing the most accurate classifier for all possible data sets.

The results of ADATE and its high average rank are rather surprising considering the severe bugs in the synthesis algorithm of the ADATE version utilized for these data sets. These bugs prevented ADATE from generating and modifying constants cleverly, affecting the data sets with continuous attributes negatively.

The only data set not affected by these bugs was dna. Surprisingly, the rank for dna is not higher than the ranks for the other data sets, and the ranks for six of the other data sets are higher. This difference might be a result of the other classification algorithms performing exceptionally well on data sets with only nominal attributes. Regardless, it shows the robustness

Table 4.2: Ranks of the classification algorithms across 10 real world data sets.

Methods	all	cmc	dna	ger	pag	sat	seg	spa	veh	yea	Avg
LMT	14	7.5	1.5	8.5	10	13	9	8	1	2	7.5
ADATE	13	1	9	5	5.5	11.5	2	11.5	8	8	7.5
M5R	6.5	3.5	15	14.5	5.5	9.5	6	6	5	5	7.7
AdaBoost	10	17.5	12	13	2	3	1	1	6	21.5	8.7
SLogistic	20	7.5	1.5	8.5	12	15	9	8	4	2	8.8
M5P	2.5	3.5	10	16.5	8.5	8	9	11.5	7	12.5	8.9
RF	16	12	22	21	2	2	3	2	13	6	9.9
Bagging	4.5	15	14	18	2	6	15.5	4	11.5	12.5	10.3
JRip	10	5.5	8	11.5	8.5	14	19	8	21.5	2	10.8
MP	18.5	9	12	11.5	20	7	18	5	3	11	11.5
ADTree	10	2	5.5	29	5.5	17.5	15.5	10	14	9.5	11.9
Logistic	17	10	24	5	14	17.5	15.5	13	2	4	12.2
REPTree	2.5	16	23	14.5	5.5	16	22	16.5	9.5	16	14.2
J4.8	1	11	17	10	16	20	13	15	17	24	14.4
NBTree	4.5	21	3.5	7	15	24	20	19	16	18	14.8
PART	10	17.5	20.5	23	12	21	12	3	18	20	15.7
SMO	25	13.5	16	5	27	11.5	21	16.5	11.5	16	16.3
LinReg	25	13.5	7	2	26	27	25	24	9.5	16	17.5
K*	21	23	27	26	12	1	7	21	15	23	17.6
DT	10	5.5	18	30	17	23	23	14	21.5	19	18.1
Ridor	6.5	22	25	19	19	19	15.5	18	23	27	19.4
RBFN	18.5	24	12	16.5	23	22	24	25	24	7	19.6
NNge	15	32	26	21	18	9.5	11	20	26	21.5	20.0
LWL	31	19.5	5.5	1	30	25	26	26	25	14	20.3
NB	32	19.5	3.5	3	29	26	27	27	29	9.5	20.6
IBk	25	29.5	28	31.5	21	4.5	4.5	22.5	19.5	25.5	21.2-
IB1	25	29.5	29	31.5	22	4.5	4.5	22.5	19.5	25.5	21.4-
VFI	33	31	20.5	21	33	28	28	31	27	28	28.1-
OneR	25	25	30	33	24	29	30	29	28	31	28.4-
VP	29	27	19	26	31	32	31	33	32	30	29.0-
DS	25	33	31	26	25	31	32	28	30	29	29.0-
HP	30	27	32.5	26	28	30	29	30	31	32	29.6-
ZeroR	25	27	32.5	26	32	33	33	32	33	33	30.7-

Table 4.3: Error percents of the classification algorithms across 10 real world data sets.

Methods	all	cmc	dna	ger	pag	sat	seg	spa	veh	yea	Avg
LMT	1.7	46.3	4.8	27.8	3.6	13	5.5	7.7	18.9	41.5	17.1
ADATE	1.6	44.3	5.7	27.4	3.3	12.9	4.4	8.1	26.0	42.9	17.7
M5R	1.4	45.1	7.3	29.4	3.3	12.7	4.8	7.6	24.6	42.2	17.8
AdaBoost	1.5	50.0	6.0	29.0	3.1	10.2	4.0	5.6	25.3	47.4	18.2
SLogistic	3.2	46.3	4.8	27.8	3.7	14.1	5.5	7.7	24.3	41.5	17.9
M5P	1.2	45.1	5.8	29.6	3.5	12.0	5.5	8.1	25.8	43.5	18.0
RF	2.1	48.8	10.7	30.6	3.1	10.1	4.6	6.1	30.3	42.3	18.9
Bagging	1.3	49.2	7.1	29.8	3.1	10.7	6.0	6.7	29.8	43.5	18.7
JRip	1.5	45.7	5.4	28.8	3.5	13.6	6.9	7.7	34.5	41.5	18.9
MP	3.1	47.6	6.0	28.8	4.8	11.3	6.5	7.2	23.6	43.4	18.2
ADTree	1.5	44.8	5.1	31.8	3.3	14.5	6.0	8.0	30.5	43.3	18.9
Logistic	3.0	48.0	12.2	27.4	3.8	14.5	6.0	8.3	21.5	41.6	18.6
REPTree	1.2	49.9	11.7	29.4	3.3	14.3	8.2	9.3	28.6	44.2	20.0
J4.8	1.1	48.6	7.9	28.0	4.1	15.1	5.9	9.0	31.9	50.4	20.2
NBTree	1.3	50.3	5.0	27.6	4.0	17.7	7.1	10	31.4	44.6	19.9
PART	1.5	50.0	9.4	31.0	3.7	15.2	5.7	6.6	33.3	45.8	20.2
SMO	3.6	49.0	7.8	27.4	8.2	12.9	8.1	9.3	29.8	44.2	20.0
LinReg	3.6	49.0	5.2	26.6	8.1	23.3	18.7	11.4	28.6	44.2	21.9
K*	3.5	51.0	23.5	31.6	3.7	9.6	5.2	11.0	30.7	49.3	21.9
DT	1.5	45.7	9.0	32.2	4.3	17.3	8.5	8.9	34.5	44.7	20.7
Ridor	1.4	50.8	12.9	30.0	4.6	14.8	6.0	9.8	35.5	51.1	21.7
RBFN	3.1	51.2	6.0	29.6	6.0	16.4	12.6	13.9	35.7	42.5	21.7
NNge	1.8	55.4	13.7	30.6	4.4	12.7	5.6	10.2	40.7	47.4	22.3
LWL	6.8	50.1	5.1	25.8	10.0	18.4	20.2	18.7	39.0	43.8	23.8
NB	6.9	50.1	5.0	27.0	9.4	20.1	21.5	20.1	58.4	43.3	26.2
IBk	3.6	55.2	24.3	32.8	4.9	10.3	4.7	11.3	34.0	50.7	23.2
IB1	3.6	55.2	26.5	32.8	5.2	10.3	4.7	11.3	34.0	50.7	23.4
VFI	9.2	55.3	9.4	30.6	11.8	27.6	24.1	26.1	48.5	53.6	29.6
OneR	3.6	53.3	36.9	33.6	6.7	40.9	36.7	22.0	50.4	60.4	34.5
VP	3.8	54.3	9.2	31.6	10.5	56.8	54.5	49.2	69.5	59.7	39.9
DS	3.6	59.8	38.6	31.6	7.0	55.5	73.5	21.7	61.9	58.1	41.1
HP	4.1	54.3	48.3	31.6	9.2	50.7	27.4	25.1	67.1	65.8	38.4
ZeroR	3.6	54.3	48.3	31.6	10.8	77.4	86.4	40.9	78.7	67.7	50.0

of the population-based search utilized by ADATE, which is able to find competitive classifiers even though there are severe bugs in the synthesis algorithm.

4.5 Discussion

Although the accuracy of the induced classifiers is an important factor in deciding which classification algorithm to use for a specific domain, it is not the only factor and there might be other factors more important depending on the situation.

In the next sections, some of these other factors are covered, and ADATE is compared to other classification algorithms according to these factors. This comparison is not complete, and its purpose is first and foremost to highlight some of the advantages and disadvantages of using ADATE to synthesize classifiers.

4.5.1 Readability

The readability of classifiers is important if there is a need for examination of the classifiers produced by a classification algorithm. Such examination may help to get a deeper understanding of the problem domain, but it may also be used to assess the usefulness of the induced classifiers.

The classification algorithms utilized in this thesis create classifiers that vary great in readability with decision trees and rule sets at one end and artificial neural networks and SVMs at the other end. Decision trees and rule sets are highly readable and naturally intuitive to humans, especially rule sets, which remain highly readable despite size increase. Artificial neural networks and SVMs, on the other hand, are complex mathematical models that are hard to understand and often nearly impossible. Thus, it is difficult to avoid "black box effects" of treating these classifiers as black boxes that magically predict the correct class.

ADATE synthesize programs that fall somewhere in between these two extremes in terms of the readability. A synthesized program can be as readable as decision trees since it is possible to represent decision trees in ADATE-ML by nesting case expressions inside case expressions, inspecting one attribute at a time until a class can be returned. This is, however, probably the simplest and most readable program that can be generated and more complex programs can be much harder to interpret, especially if advanced constructs like recursion and auxiliary functions are used. Accordingly, it might be hard to avoid "black box effects".

4.5.2 Flexibility

The flexibility of a classification algorithm relates to the degree the algorithm and the classifier produced can be configured to the problem at hand. High flexibility allows extensive customization of the classification algorithm for a specific data set in order to produce the best possible classifier. However, customization is hard, and it is not always apparent which impact the different changes will have on the classifier produced, often resulting in time consuming tuning. Thus, in some circumstances, it might be preferable to use a stringent algorithm that just works.

Flexibility and readability seem to be somewhat related, where algorithms producing simple and readable classifiers are less flexible than algorithms producing more complex and less readable classifiers. For instance, there is little flexibility in traditional decision tree learners where configuration is mostly limited to the amount of pruning employed, while there is great flexibility in neural network algorithms where the structure of the network and how the weights are trained can be configured.

Compared to the classification algorithms used in this thesis, ADATE is probably the most flexible. For example, it allows the problem domain to be accurately represented using algebraic data types and functions. Even though ADATE is flexible, writing specifications and executing ADATE do not have to be inherently hard since the flexibility can be utilized as needed. First by producing, as in this thesis, a general specification using C5conv and then gradually adding specific help functions and data types appropriate for the domain.

4.5.3 Computational Requirements

The computational requirements of ADATE is much higher than the requirements of any of the other classification algorithms executed in this thesis. For instance, when ADATE was compared empirically with other classification algorithms in section 4.4, ADATE was executed on a cluster composed of 40 cores for approximately three weeks, while the other 32 classification algorithms finished inducing the classifiers in little over two hours on a laptop with a single core Pentium M 2 GHz CPU. However, contrary to the other algorithms, ADATE searches indefinitely, which eventually may lead to discoveries impossible to make with the other algorithms.

Even though ADATE needs vast computational resources in training, the synthesized programs are fast to execute and comparable to classifiers produced by other classification algorithms.

4.6 Summary

In this chapter, we have shown how ADATE can be applied directly to classification problems and synthesize classifiers in the form of programs. Although the version of ADATE utilized had severe bugs, the synthesized classifiers proved to be competitive with the classifiers produced by state-of-the-art classification algorithms. However, considering the long execution time, it is probably wise to investigate whether other simpler classification algorithms are sufficient before utilizing the flexibility and the unique search provided by ADATE.

The next chapter shows how ADATE can be used to improve existing classification algorithms such as decision tree learning. This problem probably fits ADATE better considering the concrete need for auxiliary functions and recursion.

Chapter 5

Improving Decision Tree Learning 1

There are many types of existing classifier inducers that would be possible to improve through automatic programming, but decision tree inducers have at least three beneficial properties, making them more suitable than other algorithms. First, decision tree inducers are heavily based on heuristics, suggesting there is room for improvements. Since they have been very thoroughly researched, any generally valid improvements are a testament to the benefits of an automated experimental approach and the limitations of the human brain.

Second, the algorithmic complexity of decision tree inducers is relatively low and they are reasonably fast in both training and classification, which reduces the overall computational requirement for meta-learning through automatic programming.

Third, decision trees have a central place in numerous other classification algorithms. For instance, there are rule inducers utilizing decision trees, model tree inducers that create decision trees with models at the leaves, and decision trees are often preferred in ensemble algorithms like bagging and boosting [47]. As a result, potential improvements to a decision tree algorithm could be incorporated into other algorithms as well.

As explained in section 2.2.4, decision tree learning is composed of building and pruning, where it seems pruning may have a greater potential for improvement than building. Breiman et al. observed relatively little difference between the type of decision tree produced when different methods for selecting the most promising attribute were used, and found it more important to choose the right pruning algorithm [18]. Therefore, we chose to improve decision tree pruning.

This chapter first gives a description of pruning and three of the most

popular pruning algorithms. Next, the specification developed for improving decision tree pruning are explained along with the resulting synthesized pruning algorithm. Lastly, this algorithm is evaluated empirically on synthetic and real world data sets.

5.1 Pruning

The biggest problem in classifier induction is to overly adapt the classifier to specific details of the training set. In order to avoid such overfitting, decision tree learning employs pruning, which, as the name suggests, involves simplification of a decision tree by eliminating seemingly overfitted branches. There are two approaches to pruning, called pre-pruning and post-pruning.

Pre-pruning takes place during tree induction and tries to stop the induction process as soon as overfitting occurs. Thus, it avoids wasting resources on inducing parts of the tree that never will be used. However, it suffers from the horizon effect [48] in that it cannot explore the result of allowing the induction process to continue, making premature and overdue stopping hard to avoid [18].

Post-pruning, on the other hand, takes place after the induction process has ended and tries to simplify the tree by removing superfluous nodes. In this way, it wastes more resources than pre-pruning, yet, at the same time, it is more robust in simplification of decision trees because it has access to the full tree, allowing thorough exploration of the impact of removing nodes. Since the increased robustness normally out-weighs the increased computation, post-pruning is preferred over pre-pruning. Therefore, we will focus on improving post-pruning in this thesis.

5.1.1 Existing Pruning Algorithms

Most post-pruning algorithms are implemented quite similarly. They navigate either top-down by starting at the top of the tree and moving downwards to the leaves, or bottom-up by starting at the bottom of the tree and moving upwards to the root. During navigation, the true error rate is estimated for the different parts of the tree to decide which should be retained and which should be pruned.

Although estimation of the true error rate varies considerably between pruning methods, they normally base their estimation either on the training set or on a distinct validation set. Since the validation set is separate, the measurements from it are not biased as opposed to the measurements from the training data. This may explain the observations made by Mingers in

his comparison of pruning algorithms which lead him to conclude that the pruning algorithms utilizing a separate validation set produce more accurate trees [49].

However, in his study, Mingers did not take into account the lower amount of data available for training when a validation set is used, possibly resulting in a worse starting point. Another study performed by Esposito, Malerba and Semeraro took this into consideration and came to the opposite conclusion of Mingers [50].

In the following sections, three of the most popular pruning algorithms are explained. These vary both in navigation direction and what they base their error rate estimate on.

Reduced Error Pruning

Reduced error pruning (REP) is a simple pruning algorithm developed by Quinlan [51], which intuitively estimates the error of a tree as the number of misclassifications made when the tree is applied to a separate validation set. It traverses the tree bottom-up, and for each encountered internal node, it prunes the current node n by replacing n with a leaf predicting the majority class if it does not increase the number of errors made by the tree.

Because only the validation set is used to decide whether some parts of a tree are overfitted, it is important that the validation set is big enough so that the general patterns of the domain included in the training set also are included in the validation set. If the validation set is too small, REP might prune parts of the tree that only cover rare cases in the training set since these rare cases are not represented in the validation set. If, on the other hand, it is too large, it might result in more overfitting in the initial unpruned tree.

Pessimistic Error Pruning

Pessimistic error pruning (PEP) is a top-down algorithm designed by Quinlan [51], which for each internal node estimates and compares the error rate of pruning and retaining the current node. If the error rate of pruning the node is within one standard deviation[1] of the error rate of retaining the node, the current node is pruned and further traversal is not required. In this way, the execution time of the algorithm may be significantly lower than the execution time of other pruning algorithms that invariably traverse the whole tree.

PEP estimates the true error rate of a tree by applying the continuity correction for the binomial distribution to the error rate observed in train-

[1]The standard deviation is calculated according to the binomial distribution

ing. The use of the continuity correction has no statistical foundation, but is merely a heuristic to make the error rate observed in training more pessimistic.

Let t be a tree, $n(t)$ denote the number of instances reaching t, $e(t)$ be the number of misclassifications made by t and $c(t)$ be the children of t. Then, the error rate of a leaf is estimated as:

$$r(t) = \frac{e(t) + \frac{1}{2}}{n(t)}$$

and the error rate of an internal node is estimated as:

$$R(t) = \frac{\sum_{s \in c(t)} \left[e(s) + \frac{1}{2} \right]}{n(t)}$$

Error Based Pruning

Error based pruning (EBP) is the pruning algorithm that is used in Quinlan's popular C4.5 decision tree system [16]. It traverses the tree bottom-up while making a choice at each internal node of whether to retain the node as it is, replace it with a leaf containing the majority class, or replace the node with the most frequently used child according to the training set, a process known as tree grafting. The operation with the lowest estimated error rate is performed.

Estimation of the true error rate of a leaf is based on some questionable statistical reasoning around the error rate observed in the training set. Assuming the observed error rate is governed by a binomial distribution, the estimated error rate is the upper limit of a confidence interval with a specific confidence level, where 25% is the default. The estimated error rate of a tree is the sum of estimated error rates of the children weighted by the fraction of the instances reaching each particular child.

This algorithm is probably one of the best pruning algorithms created and it compared favorably to other pruning algorithms in [50]. Accordingly, we chose EBP as the start program for ADATE.

5.2 Specification

The following four sections describe the most important parts of the pruning specification. This specification is listed completely in appendix D. The fifth and last section, gives an overview of the alternative specifications considered.

5.2.1 User-Defined Data Types

There are five user defined data types: `class`, `split_point`, `calculated_distribution`, `c_tree_list` and `c_tree`. The first four data types contain information about the different parts of a decision tree, where `class` represents a single class value in the form of an `integer`; `split_point` corresponds to a test on a specific attribute; `calculated_distribution` contains the following information about the training instances reaching a particular node: the majority class, the number of instances of the majority class and the total number of instances; and `c_tree_list` holds a list of `c_tree`s because polymorphic lists are not allowed in ADATE-ML.

The last data type, `c_tree`, represents the decision tree itself. It can either be a decision node (`CDN`) or a leaf node (`CLeaf`). A decision node is an internal node, and contains a `split_point`, a `calculated_distribution` and a `c_tree_list` holding the children. A leaf node, on the other hand, is a terminal node and contains only a `calculated_distribution` and a `class`.

5.2.2 The f Function

The `f` function is a partial implementation of EBP, where tree grafting is omitted to keep the computational requirements low. Tree grafting is not initially expensive since the distribution is precalculated, but as soon as it is performed, the distribution must be recalculated for all the new instances reaching the subtree. Figure 5.1 presents a more readable and compact implementation written in Standard ML of the original `f` function listed in appendix D.

The `f` function takes an unpruned decision tree as input and returns a transformed decision tree along with an estimated error rate for the tree. Optimally, the `f` function should transform the unpruned decision tree to the simplest, most accurate tree possible. Nonetheless, there is nothing preventing ADATE from creating a function that adds nodes and makes the tree bigger.

Two auxiliary functions are utilized by the `f` function. The first function, `pruneCTreeList`, iterates over the trees given to it, performs pruning on them and returns a tuple containing the sum of all the returned error estimates and a `c_tree_list` with all the pruned trees.

The second function, `errorEstimate`, takes `n`, the number of instances that reach a given tree node, and `c`, the number of these instances that are correctly classified by the subtree corresponding to the node, as arguments. It calculates the upper limit error estimate of a node by approximating the binomial distribution using the normal distribution, but it performs this cal-

culation rather naively. First, the continuity correction of adding 0.5 to the number of errors is not used. Second, the normal distribution yields inappropriate values when the number of errors is either close to zero or the number of instances. For these special cases, the binomial distribution should be used directly.

We started ADATE with this "naive EBP" to see if it would be automatically improved to an algorithm competitive with EBP or even superior to it.

5.2.3 Available Functions

The functions available to ADATE during synthesis of the `f` function fall into two groups. The first group comprises constructors and functions for creation and manipulation of the built-in data types in ADATE, including constructors for boolean values; boolean functions for comparing `real` values and constants such as `rconstLess` and `realLess`; `tor` for converting a constant to a `real`; and arithmetic functions for manipulation of `real` values like `realAdd`, `realSubtract`, `realMultiply`, `realDivide`, `sqrt`, `tanh` and `ln`.

The second group contains constructors for the user-defined data types `c_tree_list`, `calculated_distribution` and `c_tree`. These constructors allow structural changes to trees, which is required to enable any form of pruning. On the other hand, the constructors for the `class` and `split_point` data types are excluded since they can easily be used to construct ill-defined decision trees by producing non-existing classes and split points.

5.2.4 Inputs and Output Evaluation Function

In this paper, an ADATE training input consists of a decision tree to be pruned. Associated with each tree is an unique synthetic data set which is split into a training partition and a test partition. Each decision tree was generated by ID3 using the training partition before starting ADATE. When ADATE is running, each pruned tree returned by the `f` function is tested on its associated decision tree test partition. The sum of the classification errors that all the pruned trees makes on their corresponding test partition determines how good the `f` function is and is used as a heuristic evaluation function that guides the ADATE search for better pruning algorithms.

There does not seem to be a correspondence between the type of decision tree learner used and the pruning algorithm [49], so any synthesized pruning algorithm working well with ID3 should work just as well with any other decision tree learner.

```
fun pruneCTreeList CTreeListNil = (0.0, CTreeListNil)
| pruneCTreeList( CTreeListCons( x, xs ) ) =
let
   val (prunedError, prunedTree) = f( x )
   val ( prunedErrors, prunedTrees ) = pruneCTreeList( xs )
in
  (prunedError + prunedErrors, CTreeListCons( prunedTree,
     prunedTrees ))
end
and errorEstimate((c, n): real * real): real =
let
  val e = (n - c) / n
  val z = 0.69
  val z2 = z * z
  val val1 = ((e / n) - ((e * e) / n )) + (z2 / (4.0 * n * n))
  val val2 = (e + z2 / (2.0 * n)) + z * (sqrt val1)
  val val3 = 1.0 + z2 / n
in
  val2 / val3
end
and f(curTree as CLeaf(
      CalculatedDist( _, numInstMajorityClass, num
    ), _ )) =
    ( num * errorEstimate(numInstMajorityClass, num), curTree )
| f ( CDN(splitPoint, dist, children) ) =
let
  val CalculatedDist( class, numInstMajorityClass, num ) = dist
  val (childErrorOnlyMultiplied, prunedChildren) =
     pruneCTreeList(children)
  val childError = childErrorOnlyMultiplied / num
  val error = errorEstimate(numInstMajorityClass, num)
in
   if childError < error then
     (num * childError, CDN(splitPoint, dist, prunedChildren))
    else
     (num * error, CLeaf(dist, class))
end
```

Figure 5.1: The initial `f` function, which is a partial and naive implementation of EBP. The implementation is rewritten in Standard ML to make it easier to read.

A synthetic data set is generated as follows. First, we create a two-layer feed forward neural network. This network always has exactly two output nodes but we chose to vary the number of input nodes, which equals the number of input attributes, between 10 and 12. We only use 0 or 1 as network inputs and also a binary target attribute, the value of which is given by the output node that has the greatest value for the given neural net inputs. The neural net weights are chosen randomly between -0.5 and 0.5. Thus, each output node is simply a random linear combination of the input nodes. A complete synthetic data set is obtained by feeding the network with all 2^n inputs for a given number n of input attributes.

As mentioned above, each data set is divided into a training partition and a test partition. The test partition is exactly half of the original data set, while we vary the portion of the other half utilized for training from 0.2 to 0.35 in steps of 0.05. In order to emulate how certain information often is missing from real world data sets, some attributes are removed completely from the data set. The number of attributes removed is varied between 0 and 4. This removal of attributes is an attempt to model the indeterminism, caused by not having sufficiently many sufficiently well chosen attributes, that is encountered in most real world data sets.

By using all combinations of the number of input attributes, the fraction used for training and the number of removed attributes, 60 different data sets are created. This procedure is repeated many times to create larger numbers of ADATE training, validation and test inputs, which all are multiples of 60 in this paper.

5.2.5 Alternative Specifications

Before deciding on this implementation of the specification, two alternative specifications were considered. The first of these two specifications was basically identical to the final specification except it used the auxiliary function `errorEstimate` as `f` function and moved the pruning algorithm inside the `main` function. In this way, it is more restrictive in terms of the amount of modification possible to the pruning algorithm, which can both be an advantage and disadvantage depending on the need to make changes outside the `errorEstimate` function.

The second specification also used the `errorEstimate` function as `f` function, but it utilized another method for testing the performance of the synthesized programs. Instead of representing the problem as pruning an unpruned tree, the problem is simplified to only concern whether pruning should be employed at a particular node. Thus, each input is a tuple holding the distribution for the current node and the distributions for its children.

In order to evaluate these specifications, we executed them in parallel on a cluster for approximately a week. We compared them both in performance and the degree of overfitting in the population. The programs produced according to the final specification performed the best, and there seemed to be less overfitting in the population compared to the other two populations. Of the other two specifications, there seemed to be more overfitting in the population for the specification with the simplified evaluation scheme, possibly contributed by the fact that the simplification makes it easier for a program to perform well by chance.

5.3 Execution of ADATE

The ADATE system was executed on a cluster with 16 computers over a period of three weeks. The 16 computers have a total of 22 CPU cores that are a mix of Intel Pentium D and Pentium 4.

Before performing this execution, we conducted a set of small experiments in order to try to determine the optimal size of the training set with respect to both overfitting and synthesis speed. The size should be small to obtain a short execution time, but at the same time it should be big enough to avoid excessive overfitting. Three specifications with 60, 120 and 240 instances were executed, and the 120 and 240 specifications yielded the most accurate programs on a separate synthetic validation set with 1200 instances. Although the 120 specification did not show any signs of excessive overfitting, the danger of eventual overfitting when more execution time is allowed lead us to choose 240.

During execution of ADATE, the training size was increased to 480 and eventually to 960 as we observed some signs of overfitting by studying how the classification accuracy on the validation set increased with program size.

5.3.1 Selecting the Pruning Algorithm

The ADATE population contains many programs at any given time. Therefore, some method must be applied to finally select the program that is likely to produce the most accurate decision trees in general.

Since more computation time is available per program during this final selection, we chose to use a larger validation set with 15000 instances and selected the program with the total least number of classification errors on this set.

The selected pruning algorithm has a structure similar to that of the initial pruning algorithm and the logic of the tree traversal algorithm was

```
fun pruneCTreeList CTreeListNil = (0.4026881, CTreeListNil)
| pruneCTreeList CTreeListCons( x, xs ) =
let
   val (prunedError, prunedTree) = f( x )
   val ( prunedErrors, prunedTrees ) = pruneCTreeList( xs )
in
  (prunedError + prunedErrors,
    CTreeListCons( prunedTree, prunedTrees ))
end
```

Figure 5.2: The `pruneCTreeList` function of the synthesized pruning algorithm produced by ADATE.

```
fun errorEstimate( c, n ) =
let
    val v1 = tanh( tanh( tanh( (n - c) /  n ) ) )
    val v2 = sqrt( tanh( tanh( sqrt( n ) ) ) )
    val v3 = tanh( sqrt( sqrt( c ) ) )
in
    (v1 + v2) / v3
end
```

Figure 5.3: The `errorEstimate` function of the synthesized pruning algorithm produced by ADATE.

not changed by ADATE. This pruning algorithm is given in its original form in appendix E.

However, ADATE modified the initial auxiliary functions `pruneCTreeList` and `errorEstimate`. We rewrote these two functions in Standard ML where we changed the ADATE generated variable names to make the functions more readable and compact. The `pruneCTreeList` is presented in figure 5.2 and it is almost identical to its initial version, but instead of starting from zero when summing the error estimates, it starts from 0.403. This increase in error might increase the amount of pruning performed.

The `errorEstimate` function generated by ADATE is different from anything we have seen and still apparently quite practically useful for pruning. It is given in Figure 5.3 and consists of an arithmetic expression that is somewhat hard to interpret.

Note that the `tanh` function, being a scaled version of a sigmoid function, often is used in the nodes of feed forward neural nets and that weight optimization for such nets typically results in a "black box effect" that makes the nets hard to analyze in detail. Unfortunately, the automatically synthesized

function in Figure 5.3 also suffers from this black box effect.

Thus, we may notice experimentally that this ADATE generated function results in better pruning than the confidence interval upper limit estimation used by the EBP of C4.5, but it is hard to analyze it theoretically.

5.4 Results and Analysis

The synthesized pruning algorithm (SYNTH), which was partially shown in Figures 5.2 and 5.3, is evaluated empirically across both synthetic and real world data sets by comparing it with four other pruning algorithms, namely no pruning (NOP), the naive implementation of EBP (NEBP) that ADATE used as its initial program, EBP without tree grafting (WEBP) and EBP without tree grafting(TWEBP), where the confidence level is tuned using 10 fold cross-validation. In addition, it is compared to 31 of the 32 classification algorithms utilized in section 4.4 when empirically evaluating synthesized classifiers. VFI is not used since it was not able to produce a classifier for all the data sets.

The algorithms are compared over multiple data sets using the Friedman test as in section 4.4, and it will try to reject the null hypothesis of all algorithms being equally accurate. If the null hypothesis is rejected at significance level 0.05, the Holm post hoc test is employed to investigate which of the algorithms that are truly different from SYNTH.

5.4.1 Synthetic Data

The synthetic data sets for testing are generated using the same procedure that was employed when creating the training and validation sets for ADATE, but the procedure is repeated many more times. We used 1000 repetitions when comparing SYNTH with other pruning algorithms, resulting in 60000 new data sets with 12000 data sets for each specific number of attributes to remove. In this way, it is possible to determine with statistical significance whether the synthesized pruning algorithm produces more accurate trees in this domain compared to the other pruning algorithms.

However, we used only 16 repetitions to generate the data sets when SYNTH is compared with other classification algorithms in order to keep the execution computationally feasible. This yielded a total of 960 data sets with 196 data sets for each specific number of attributes to remove.

Table 5.1: Average ranks of the pruning algorithms on synthetic data.

#Removed Attribs	#Data sets	NOP	NEBP	WEBP	TWEBP	SYNTH
0	12000	3.426-	2.936-	3.078-	3.010-	2.549
1	12000	3.885-	3.318-	2.618-	2.632-	2.548
2	12000	4.245-	3.532-	2.360+	2.395+	2.469
3	12000	4.445-	3.534-	2.278+	2.360	2.384
4	12000	4.483-	3.330-	2.349+	2.440-	2.398
Total	60000	4.097-	3.330-	2.536-	2.567-	2.470

Pruning Algorithms

Table 5.1 contains the average rank of the pruning algorithms for each of the data sets where a specific number of attributes are removed. The last row shows the average rank of the algorithms for all the 60000 data sets. A + means the algorithm is significantly better than SYNTH, and a - means it is significantly worse.

The ADATE generated SYNTH pruning performs overall better than all the other pruning algorithms. It has a rank considerably higher than that of NOP and NEBP, both in total and for each type of data sets where a specific number of attributes are removed.

This shows that the ADATE system has been able to significantly improve the initial pruning algorithm, NEBP, for this domain. On the other hand, the difference in rank is not that great compared to WEBP and TWEBP, and it has a somewhat lower average rank than that of WEBP for the data sets where two, three, and four attributes are removed. However, SYNTH performs so much better on the data sets where no attributes are removed that it overall produces the most accurate decision trees.

A surprising observation is that WEBP has a slightly higher rank than that of TWEBP on average. This is probably caused by the limited number of training instances for some of the data sets. For instance, the smallest training set contains only 102 training instances.

Other Classification Algorithms

Table 5.2 contains the average rank of the 5 pruning algorithms and the 31 classification algorithms for the data sets where a specific number of attributes are removed. The last column includes the average rank of the algorithms across all the 960 data sets, and the algorithms are ordered according to this column. A + means the algorithm is significantly better than

SYNTH, and a - means it is significantly worse.

SYNTH is worse than 16 of the 36 algorithms and it is outperformed by 15 of these algorithms. It has an average rank of 18.2, which is considerably lower than 6.3 for the best ranked algorithm, Simple Logistic.

Still, these results are not surprising considering the underlying model is basically a linear combination of the attributes, and there is a clear tendency in the type of algorithms that perform well. 11 of the 15 algorithms make use of functional models in some way, 3 of the algorithms are related to naive bayes and only one algorithm is actually using traditional decision trees with classes at the leaves. Accordingly, the inductive bias of traditional decision tree learners is probably not suited for these data sets, and it is unrealistic to expect ADATE to modify the inductive bias of decision tree learning to such an extent that SYNTH becomes significantly better than other classification algorithms for these data sets.

Despite being outperformed by 15 algorithms, SYNTH is far from the worst algorithm and outperforms a collection of 17 decision tree learners, rule learners and instance-based methods. This include two of the pruning algorithms previously compared to SYNTH and the results resemble the previous results, where NOP and NEBP differ vastly in rank compared to SYNTH.

The last two pruning algorithms, TWEBP and WEBP, on the other hand, are no longer found to be significantly worse than SYNTH, and WEBP is slightly better than SYNTH, although not significantly. This shows that SYNTH, TWEBP and WEBP perform almost the same, and it is only possible to statistically separate them by utilizing a great number of data sets such as 60000.

5.4.2 Real World Data

The 14 data sets are taken from the UCI machine learning repository [41]. They have only discrete attributes since continuous attributes are not supported by the decision tree inducer, ID3, learning the initial unpruned decision trees. In addition, missing values are not supported by ID3, so any missing values are replaced with the most frequent attribute value.

Data Sets

The specific properties of the data sets are presented in table 5.3.

Agaricus-Lepiota (aga) Jeff Schlimmer contributed this data set, and it contains 8124 different mushrooms belonging to the Agaricus and Lepiota

Table 5.2: Average ranks of five pruning algorithms and 31 other classification algorithms across synthetic data sets.

Methods	0	1	2	3	4	Avg
SLogistic	4+	5.5+	5.6+	7.6+	8.9+	6.3+
Logistic	4.4+	6+	5.6+	7.3+	8.8+	6.4+
LMT	4.1+	5.7+	5.9+	7.8+	9.3+	6.6+
SMO	6.3+	7.2+	6.9+	9.5+	11.1+	8.2+
M5P	10.2+	8.8+	8+	8+	9.5+	8.9+
M5Rules	10.4+	8.6+	8.6+	8.7+	9.6+	9.2+
LinReg	12.9+	11.4+	9.3+	10.8+	9.9+	10.8+
NB	14.9+	12+	11+	10.3+	10.4+	11.7+
VP	12.8+	12.6+	11.1+	11.5+	11.1+	11.8+
ADTree	12.4+	11.5+	12.1+	11.2+	13.5+	12.1+
NBTree	14.6+	15.3+	15.4	14.4+	13.5+	14.7+
LWL	17.4	15.4+	13.3+	13+	12.1+	14.2+
RBFN	14.6+	15.7+	15.4	15.9	17.7	15.9+
Bagging	17.2	16.1	15.9	16.3	15.8	16.3+
MP	5.5+	13.8+	18.9	21.9-	23.6-	16.7+
WEBP	20.1	19	18.1	17.4	16.2	18.1
SYNTH	19.3	18.9	18.3	17.7	16.6	18.2
J48	20.3	18.9	18.7	17.2	16.5	18.3
TWEBP	20.4	18.9	18.2	18.3	17.2	18.6
PART	18.4	19.2	20.8	21.1-	22-	20.3-
AdaBoost	14.6+	18.5	21.5-	23.4-	24.1-	20.4-
DT	24.3-	21.8	18.9	19.5	19	20.7-
JRip	24.5-	22.1-	21.4-	20.1	18.4	21.3-
REPTree	26.4-	22.9-	21.5-	18.8	19.5	21.8-
RF	16.8	18.7	22-	24.7-	27.2-	21.9-
NEBP	19.8	23.1-	24.2-	23.9-	20.9-	22.4-
K*	25.3-	24.3-	22.8-	21.5-	20.4-	22.8-
Ridor	25.3-	23.2-	22.9-	22.3-	20.2-	22.8-
NNge	17.8	20.9	24.5-	27.7-	30.1-	24.2-
NOP	21.1	25.3-	26.4-	28.4-	28.3-	25.9-
IBk	28.8-	27.8-	26.7-	25.8-	26.9-	27.2-
OneR	32-	30.9-	29.8-	26.6-	24.8-	28.8-
DS	32-	30.9-	29.9-	26.8-	24.7-	28.8-
ZeroR	32.6-	31.7-	31-	27.9-	26.1-	29.9-
IB1	31.4-	31.1-	31.7-	31.6-	31.2-	31.4-
HP	33.2-	32.7-	33.9-	31.1-	30.8-	32.3-

Table 5.3: Properties of the 14 nominal data sets.

Name	# Instances	# Attributes	Missing	# Classes
aga	8124	22	yes	2
aud	226	69	yes	24
car	1728	6	no	4
dna	3186	60	no	3
hay	160	4	no	3
krkp	3196	36	no	2
mon1	556	6	no	2
mon2	601	6	no	2
mon3	554	6	no	2
nur	12960	8	no	5
pro	106	57	no	2
soy	683	35	yes	19
tic	958	9	no	2
vot	435	16	no	2

Family. The physical properties of each mushroom are described through 22 nominal attributes, and the objective is to determine whether a mushroom is poisonous or editable.

Audiology (aud) This data set was donated by Ross Quinlan after standardizing the attributes in the original data set contributed by Professor Jergen at Baylor College of Medicine. The problem is to diagnose a patient in the field of audiology by considering 69 nominal attributes. There are 226 instances, and 24 different diagnoses to choose between.

Car (car) The car data set was contributed by Marko Bohanecw, where the objective is to evaluate a car based on six attributes representing properties like price, comfort and safety. There are 4 different classes and a total of 1728 instances.

Splice-junction Gene Sequences Database (dna) Dna has been previously explained in section 4.1

Hayes-roth (hay) This database was created by Barbara and Frederick Hayes-Roth and contributed by David W. Aha. It contains 160 instances, 4 nominal attributes and 3 different classes.

King-rook vs King-pawn (krkp) This data set donated by Rob Holte consists of 3196 chess end-games, where the white player has the king and rook left and the black player has the king and pawn left. The pawn is located in square A7 and it is the white player's turn. The objective is to determine whether the white player can win considering the position of the pieces described in terms of 36 nominal attributes.

Monks-1 (mon1) This data set was artificially created and donated by Sebastian Thrun at Carnegie Mellon University. The objective is to determine whether a robot described by six nominal attributes is in a specific group or not. An underlying rule, different for each of the three monk data sets, determines which of the robots that are members of this group. There are a total of 556 instances.

Monks-2 (mon2) Monks-2 has similar properties as Monks-1, but it has 601 instances.

Monks-3 (mon3) This data set is generated similarly to Monks-1 and Monks-2 except 5% noise is injected to the classes. It has a total of 554 instances.

Nursery (nur) Marko Bohanec and Blaz Zupan donated this data set, and it is created from an underlying hierarchical decision model used in Slovenia to objectively assess applications for nursery school. Each application is structured into three groups: employment, family structure and finance, and social and health of the family. These groups have eight underlying attributes that are included in the data set. There are 5 different classes and 12960 instances.

Promoters (pro) This data set was donated by M. Noordewier and J. Shavlik, and the aim is to determine whether a DNA sequence is a E. Coli promoter. Each DNA sequence has 57 base-pairs, and there are 106 instances in the data set.

Soybean (soy) Ming Tan and Jeff Schlimmer contributed this data base, where the problem is to diagnose the type of soybean disease according to a set of symptoms encoded in 35 attributes. There are 19 different classes and 183 instances.

Table 5.4: Ranks of each pruning algorithm on real world data sets. Obviously, smaller numbers are better.

Data sets	NOP	NEBP	WEBP	TWEBP	SYNTH
aga	3(0)	3(0)	3(0)	3(0)	3(0)
aud	2(24.5)	2(24.5)	4(28.4)	5(28.9)	2(24.5)
car	2.5(5.6)	2.5(5.6)	5(6.3)	4(5.7)	1(5.4)
dna	5(8.4)	4(7.3)	1.5(6.1)	1.5(6.1)	3(6.9)
hay	4.5(28.8)	4.5(28.8)	1.5(27.5)	1.5(27.5)	3(28.1)
krkp	1.5(0.3)	1.5(0.3)	5(0.5)	3.5(0.4)	3.5(0.4)
mon1	3(2.2)	3(2.2)	5(2.9)	1(2.0)	3(2.2)
mon2	1.5(29.3)	1.5(29.3)	4.5(34.3)	4.5(34.3)	3(30.1)
mon3	5(2.7)	2.5(1.1)	2.5(1.1)	2.5(1.1)	2.5(1.1)
nur	1(1.2)	2.5(1.5)	5(3.2)	4(2.4)	2.5(1.5)
pro	2(25.2)	3(25.3)	5(30.3)	4(26.4)	1(23.5)
soy	4(7.9)	2.5(7.8)	2.5(7.8)	5(8.1)	1(7.5)
tic	4.5(14.9)	4.5(14.9)	2.5(14.8)	1(13.9)	2.5(14.8)
vot	5(6.7)	4(6.4)	2(4.8)	1(4.4)	3(5.1)
Avg	3.180(11.3)	2.929(11.1)	3.500(12.0)	2.964(11.5)	2.430(10.8)

Tic Tac Toe (tic) This data set supplied by David W. Aha contains 958 different tic tac toe end-games, where the task is to determine whether player 1, who started, has won. There are 9 attributes, one for each cell containing x for player 1, o for player 2 and b for blank.

House Votes (vot) This database was donated by Jeff Schlimmer and contains voting records from 1984 United States Congressional Voting Records Database. The problem is to determine whether a congress man is either a democrat or republican according to how this person voted in 16 key areas. There are 435 instances in total.

Results

Table 5.4 shows the ranks given to each pruning algorithm for the different data sets according to the average error percent found using 10 fold cross-validation. This error percent is shown in parenthesis. The last row includes the average rank and error percent of the algorithms across all data sets.

SYNTH performs better than the other pruning algorithms with a total average rank of 2.4 compared to 2.9 for the best of the others. Still, there is not enough statistical evidence to conclude that some of the algorithms are truly different from one another for these real world data sets and the null

Table 5.5: Average number of nodes in the pruned trees for the pruning algorithms on real world data sets.

Data sets	NOP	NEBP	WEBP	TWEBP	SYNTH
aga	38.0	38.0	38.0	38.0	38.0
aud	136.5	136.5	79.4	83.2	131.4
car	387.6	342.8	180.4	221.2	329.8
dna	720.2	449.8	181.4	172.2	393.4
hay	59.6	52.1	29.7	30.9	47.3
krkp	91.4	90.6	60.4	66.4	67.0
mon1	96.7	96.7	66.9	77.8	94.7
mon2	446.4	434.1	1.0	1.0	403.4
mon3	60.0	17.6	17.6	17.6	17.6
nur	1133	1010.9	501	727.2	879.3
pro	37.8	33.0	19.4	17.0	32.2
soy	248.1	186.1	138.7	143.4	169.9
tic	319.6	319.6	163.9	216.1	282.7
vot	66.4	54.7	13.9	4.0	29.8
Avg	274.4	233.0	106.6	129.7	208.3

hypothesis could not be rejected.

The table contains some unexpected results. First, performing no pruning for some of these data sets seems to be a good strategy and NOP has the highest rank for several of the data sets. In addition, NOP has an average error percent better than that of TWEBP and WEBP. Second, the naive version of EBP, NEBP, performs better than WEBP and TWEBP both in terms of rank and error percent.

Detailed information about the amount of pruning employed by the different algorithms is reported in table 5.5. Each row contains the average size of the pruned trees produced for the different data sets, except the last row containing the average tree sizes across all data sets.

There is a distinct pattern in the amount of pruning performed by the different algorithms compared to each other for these data sets. Naturally, NOP creates the largest trees since no pruning is employed. NEBP is similar to NOP and performs almost no pruning, while TWEBP and WEBP perform the most pruning of all the algorithms. This is rather surprising considering the close algorithmic relationship between the three algorithms, yet it is consistent with the rank and error percent of these algorithms where the amount of pruning performed by NEBP appears to be better suited for these data sets.

SYNTH prunes more than NOP and NEBP and less than WEBP and

TWEBP, a strategy that is the most appropriate on average for these data sets considering the ranks and error percents observed.

5.5 Summary

This chapter has explained how ADATE can be used to improve decision tree learning by focusing on a specific part of it, namely pruning. The pruning algorithm chosen for improvement was a partial implementation of EBP, and numerous synthetic data sets with simple two-layer neural networks as their underlying models were used to guide the synthesis.

The resulting synthesized pruning algorithm was evaluated empirically across synthetic data sets, and it was found to perform significantly better than the start program as well as the other pruning algorithms tested. The same pruning algorithms were evaluated on real world data sets, where the synthesized pruning algorithm performed the best on average, even though it was not significantly better than any of the other algorithms. Despite that, the improvements to decision tree learning contributed by ADATE seemed small, especially compared to other classification algorithms.

In the next chapter, we try to achieve greater advancements by addressing areas of the specification presented in this chapter where there are room for improvements.

Chapter 6

Improving Decision Tree Learning 2

Based on the results and discoveries from the execution of the first pruning specification, we developed a second pruning specification mostly identical to the first, but with some modifications.

This chapter first gives a description of the modifications made to the first specification described in the previous chapter. Second, an explanation is provided of how ADATE was executed, how the synthesized pruning algorithm was selected and how the synthesized pruning algorithm performs pruning. Lastly, the results from the empirical evaluation of the synthesized pruning algorithm are presented.

6.1 Specification

Since the specification remains mostly the same as the first specification, only the modifications made to the first version are described. The three most important changes include scaling of the arguments passed to the `CalculatedDist` constructor, using WEBP as the initial `f` function and generating synthetic data sets with a varying number of attribute values and classes along with a more complex underlying neural network. The whole specification is listed in appendix F.

6.1.1 User-Defined Data Types

Most of the data types remain exactly the same as in the first version except for `c_tree` and `calculated_distribution` where changes were made to the `CLeaf` and `CalculatedDist` constructors. `CLeaf` is simplified and

the `class` previously included in addition to the `calculated_distribution` is removed. This `class` is superfluous since it is already included in the `calculated_distribution`. As a result, there are fewer alternatives for ADATE to consider, possibly speeding up program synthesis.

Two changes are made to the `CalculatedDist` constructor. First, an extra argument is added holding the size of the training set, which might be useful in synthesis. Second, all the numerical arguments of `CalculatedDist` are now scaled with regard to how constants are generated in ADATE. Approximately 50% of the time is spent searching for constants in the interval -0.5 and 0.5, while the remaining 50% is spent searching outside this interval. Consequently, for the first specification, where the values typically would lie well beyond this interval, most of the time was spent generating too small constants without any practical use.

The three numerical arguments of `CalculatedDist` are scaled as follows:

$$sc = \frac{c}{2n}, sn = \frac{n}{100}, sN = \frac{5n}{N}$$

where n is the number of instances reaching a particular node, c is the number of these instances matching the majority class, N is the total number of instances and sc, sn and sN are their scaled counterparts.

6.1.2 The f Function

The `f` function has undergone some changes, but most of the changes are made to the auxiliary `errorEstimate` function. Thus for the sake of brevity, we only present a rewritten Standard ML version of `errorEstimate` in figure 6.1, while the original and the rewritten `f` function are presented in appendices F and G.

The `errorEstimate` function has been changed to correct the flaws of the original `f` function regarding, as previously explained, the omission of the continuity correction and the inappropriate use of the normal distribution when the number of errors are either close to zero or the number of instances. As a consequence of these changes, the initial `f` function is now regarded as an implementation of EBP without tree grafting. Note that the `errorEstimate` function only returns the amount that should be added to the error to get the upper limit of the confidence interval, and does not return the complete error estimate as the `errorEstimate` function in the first specification did.

Because of the changes to the `CalculatedDist` constructor, an extra argument holding the training set size has been added to the `errorEstimate` function. In addition, it unscales the scaled numbers given as input in order to make the calculation correct.

```
fun errorEstimate((sc, sn, sN): real * real * real): real =
let
  val n = sn * 100.0
  val e = n - ( sc * 2.0 * n)
in
  if e < 1.0 then
  let
    val base = n * (1.0 - pow(0.25, 1.0 / n))
  in
    if 0.0 < e then
      base + e * (
        (errorEstimate((n-1.000000001)/(2.0 * n), sn, sN)
      ) - base)
    else
      base
  end
  else
    if n < e + 0.5 then
      if 0.0 < n - e then n-e else 0.0
    else
    let
      val errorRate = (e + 0.5) / n
      val z = 0.674489751129221500

      val sq = (errorRate / n) -
            (errorRate * errorRate / n) +
            (z * z / (4.0 * n * n))

      val val1 = errorRate + (z * z) / (2.0 * n) + z * (sqrt sq)
      val val2 = (1.0 + (z * z) / n)
      val r = val1 / val2
    in
      ((r * n) - e)
    end
end
```

Figure 6.1: The auxiliary `errorEstimate` function of the initial `f` function rewritten in Standard ML.

6.1.3 Available Functions

There are one addition and three restrictions to the functions available to ADATE during synthesis. We decided to include the = function to enable comparison of classes, and restrict access to the `split_point` data type by making it abstract since there is no clear need for it in a pruning algorithm. Furthermore, we evaluated whether to include the arithmetic `ln` and `sqrt` functions by executing two specifications, one with `ln` and `sqrt` and one without, in parallel for a short amount of time. Since no distinct difference in performance was observed, we decided to discard these two functions.

6.1.4 Inputs and Output Evaluation Function

In the first specification, all the synthetic data sets had only attributes and classes with two values, meaning all the unpruned decision trees trained had only decision nodes with two branches. Since the trees had so similar properties, there was an increased danger of overfitting. Thus, in this specification, the number of attribute and class values are randomly varied between two, three and four.

The neural network corresponding to each data set is changed from a two-layer neural network to a three-layer network containing an output, an input and a hidden layer. Identically to the first specification, each class is represented as an output node, and the output node with the highest value for a given set of inputs determines the class.

However, there is no longer a one to one relationship between the input nodes and the attributes since each attribute can have more than two values. We encode the attribute values as binary numbers, meaning 0 as 0, 1 as 1, 2 as 10 and 3 as 11. Consequently, attributes with two values are represented as one node, while attributes with three or four values are represented as two nodes.

Moreover, the complexity of the network is varied by randomly selecting the number of nodes in the hidden layer between between 5 and 11. Each node in the hidden layer has a tanh activation function.

Similarly to the first specification the training fraction is varied between 0.2 and 0.35 in steps of 0.05 and the number of attributes removed is varied between zero and five, while the number of attributes is now varied between 6 and 10. By using all combinations of these parameters, a total of 100 data sets are generated and, as in the first specification, this generation procedure is repeated multiple times to create ADATE training, validation and test inputs.

6.2 Execution of ADATE

The ADATE system was executed on a cluster with 28 computers composed of a total of 42 CPU cores that are a mix of Intel Pentium D and Pentium 4. A month or more will eventually be given to the execution of ADATE, but we extracted the final program after approximately two weeks of execution in order to present preliminary results of this second specification in this thesis. Thus, the results are likely to only become better as ADATE continues to execute until termination.

Contrary to the first specification, we did not perform any initial experimentation with the number of training inputs to include before executing the ADATE system. Instead, we decided to start with only 100 training inputs and increase the number of inputs as needed during execution.

This experimental procedure is based on some rather intriguing details observed during execution of the first specification about how the synthesized programs evolved in performance relative to each other. The difference in performance between the synthesized programs in the population is great at the beginning of synthesis, and it is clear that valuable advancements are made. However, this difference decreases with time to the extent that there is almost no difference. Typically, this is a sign of overfitting in ADATE, but the programs make similar advancements on the validation inputs as on the training inputs, suggesting no overfitting.

A possible explanation for this decrease in performance improvement is that it might be more difficult to make as big improvements when approaching the optimal, achievable performance of pruning the initial unpruned trees. Figure 6.2 shows a graphical representation of this evolvement where the best program slowly approaches the optimal pruning algorithm. In this way, higher amount of data is required later in the synthesis than in the beginning to be able to spot improvements made to the programs.

By following the experimental procedure suggested above, it is possible to increase the execution speed and at the same time enable ADATE to distinguish between the programs synthesized.

6.2.1 Selecting the Pruning Algorithm

Similarly to the first specification, we used a much bigger validation set to select the final pruning algorithm, but because of the increased size of the synthetic data sets, we were forced to use 5000 instances instead of 15000. Figure 6.3 presents a rewritten version of the selected `f` function in Standard ML, while the original is presented in appendix H.

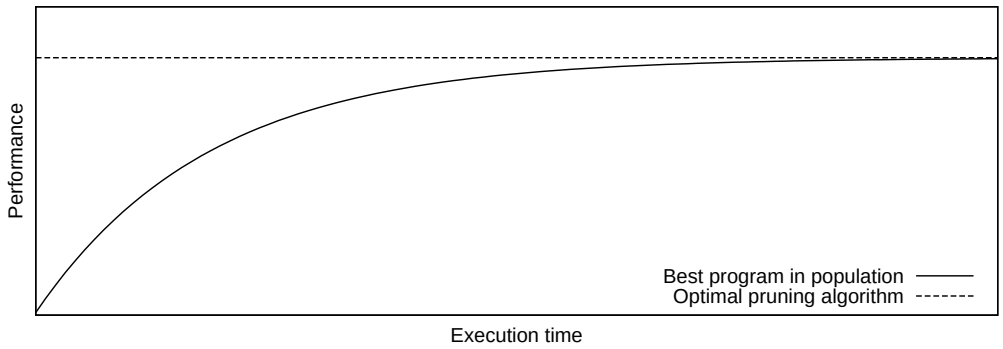

Figure 6.2: A graphical representation of how the performance of the best program seems to evolve in relation to the optimal pruning algorithm.

The selected `f` function employs the same bottom-up tree traversal algorithm as the start program and uses a modified version of the auxiliary function `pruneTreeList` to prune the children as well as to calculate their estimates. At first the modifications to the `pruneTreeList` might seem great since the current tree is made the last member of the `c_tree_list` returned. However, this has no impact on classification since the last branch does not correspond to an attribute value and will never be traversed. Thus, the `pruneTreeList` is exactly the same as the original except it adds 0.235 to the sum of the estimates of the children.

The `errorEstimate` function included in the original `f` function is removed, and it calculates the estimates somewhat differently depending on whether the nodes are a list of children or a single decision node. Note, the values calculated are estimates of the accuracy and not the error, explaining why pruning is performed if the children have a lower estimate than its parent node.

Calculation of the estimate of a set of children involves `sN` at the current node and `childError`, the sum of the estimates of the leaf nodes located in the subtree rooted at each child. The estimate of a leaf node is determined by `sn` and `sc`, where `sn` controls the minimum and maximum value returned such that the interval increases as `sn` increases, while `sc` determines the position of the final value in this interval so that a high `sc` results in a low value and the other way around. Thus, accurate leaves covering many instances receive low estimates contributing to a low `childError` which results in a high overall estimate of the children since the `childError` is subtracted.

Calculation of the estimate of an internal decision node involves only `sn` and `sc`. The value of `val1` is calculated similarly to the estimate of a leaf node except other constants are used and `sc` is not subtracted from the final

```
fun f (curTree as CLeaf( dist)) =
let
  val CalculatedDist(_, sc, sn, _) = dist
  val val0 = sn * 67.92176711838479
in
  (((val0 - (sc * 2.8584295366524395 * val0)) + sc), curTree)
end
| f ( curTree as CDN(splitPoint, dist, children) ) =
let

  fun pruneTreeList CTreeListNil = (0.2350543345568945,
    CTreeListCons( curTree, CTreeListNil ))
  | pruneTreeList ( CTreeListCons( x, xs ) ) =
  let
    val (prunedError, prunedTree) = f( x )
    val ( prunedErrors , prunedTrees ) = pruneTreeList( xs )
  in
    (prunedError + prunedErrors, CTreeListCons( prunedTree,
       prunedTrees ))
  end

  val ( childError, prunedChildren ) = pruneTreeList( children )
  val CalculatedDist(_, sc, sn, sN) = dist

  val val3 = (sn * 59.59799248750001)

  val val1 = (val3 - ((sc * 2.911831626001407) * val3))
  val estimate = sn - val1

  val val2 = (tanh(tanh(1.052447543004225 + childError) - sN) +
      sN)

  val childEstimate = (val2 - childError)
in
  if childEstimate < estimate then
    ( f( CLeaf dist ) )
  else
    ( childError, CDN( splitPoint, dist, prunedChildren ) )
end
```

Figure 6.3: The `f` function of the synthesized pruning algorithm rewritten in Standard ML for clarity and compactness.

value. Since `val1` is subtracted from `sn`, it is clear that the highest estimate is achieved by the decision node that covers the most instances belonging to the same class.

Still, it is hard to determine by only reviewing the source code when pruning is performed and whether this strategy is better than the original. However, this may be answered empirically by the results presented in the following section.

6.3 Results and Analysis

The same empirical study performed for the first decision tree pruning specification is performed for the second specification as well. As a result, the synthesized pruning algorithm, SYNTH, is compared across synthetic and real world data sets to three other pruning algorithms, namely no pruning (NOP), the initial `f` function implementing EBP without tree grafting (WEBP), and EBP without tree grafting utilizing cross-validation to tune the confidence level (TWEBP). In addition, SYNTH is compared with 31 classification algorithms from WEKA on synthetic data sets.

In order to statistically evaluate the results, the Friedman and Holm post hoc test are utilized to determine whether the algorithms are truly different at 0.05 significance and specifically which of the algorithms that are truly different from SYNTH.

6.3.1 Synthetic Data

The synthetic data sets for testing are generated using the same procedure previously described for generating ADATE training and validation inputs. When comparing SYNTH with other pruning algorithms, 60000 data sets are generated where 12000 data sets are generated for each specific number of attributes to remove. Due to the large computational requirements of evaluating the different algorithms, only 1000 data sets are utilized when comparing SYNTH with other classification algorithms where 200 data sets are created for each number of attributes removed.

Pruning Algorithms

Table 6.1 shows the average ranks of the pruning algorithms on 60000 synthetic data sets. The last row contains the average rank of the pruning algorithms for all the data sets, while the other rows contain the average rank of the pruning algorithms for the data sets where a specific number

Table 6.1: Average ranks of each pruning algorithm on synthetic data.

#Removed Attribs	#Data sets	NOP	WEBP	TWEBP	SYNTH
0	12000	3.293-	2.585-	2.148-	1.974
1	12000	3.577-	2.251-	2.138-	2.034
2	12000	3.707-	2.087	2.105	2.101
3	12000	3.615-	2.100+	2.137	2.148
4	12000	3.091-	2.303	2.284+	2.322
Avg	60000	3.457-	2.265-	2.162-	2.116

of attributes are removed. A + is included if the algorithm is significantly better than SYNTH and a - is included if it is significantly worse.

SYNTH performs significantly better on average than the other pruning algorithms with an average rank of 2.116. This rank is remarkably higher than that of NOP, and SYNTH outperforms NOP for each of the data sets where a specific number of attributes are removed.

On the other hand, the difference in average rank is not that great compared to WEBP and TWEBP. SYNTH only outperforms them when zero and one attribute are removed, while they outperform SYNTH when three and four attributes are removed. Although the difference is not that great, it shows that ADATE has been able to significantly improve the initial pruning algorithm, WEBP.

Other Classification Algorithms

Table 6.2 contains the average ranks of the 31 classification and the 4 pruning algorithms on 1000 synthetic data sets. Each column contains the average rank of the algorithms for the data sets where a specific number of attributes are removed, except for the last column containing the average ranks across all the data sets. A + or - is included if the algorithm is either significantly better or worse than SYNTH according to the Holm post hoc test.

SYNTH performs poorly on these synthetic data sets compared to the other classification algorithms with only an average rank of 19.0, which is remarkably lower than the average rank of the best algorithm, Logistic. In addition, 19 of the classification algorithms are better than SYNTH and 17 of these are significantly better.

Despite that, 14 of the algorithms outperforming SYNTH are related to mathematical or bayesian models, while the remaining three algorithms consist of the nearest neighbour algorithm, K*, and the two ensemble algorithms, bagging and AdaBoost.M1, making use of the traditional decision learner,

Table 6.2: Average ranks of 4 pruning algorithms and 31 other classification algorithms on synthetic data.

Methods	0	1	2	3	4	Avg
Logistic	4.1+	4.3+	4.8+	6.8+	8.5+	5.7+
LMT	3.8+	4.4+	5.9+	6.8+	9.1+	6.0+
SLogistic	4.6+	5.1+	5.7+	7.1+	9.2+	6.4+
M5P	7.3+	6.5+	6.0+	7.0+	9.7+	7.3+
M5Rules	7.7+	6.9+	6.3+	7.4+	9.9+	7.6+
SMO	4.8+	5.5+	6.7+	11.1+	15.3	8.7+
NB	11.7+	9.3+	8.4+	9.5+	10.7+	9.9+
LinReg	10.2+	10.1+	9.5+	10.5+	12.2+	10.5+
NBTree	11.2+	10.8+	10.2+	10.4+	10.7+	10.7+
LWL	12.2+	10.3+	10.1+	10.8+	11.5+	11.0+
ADTree	13.3+	11.8+	10.4+	10.8+	11.3+	11.5+
RBFN	12.6+	11.7+	12.8+	15.2	16.1	13.7+
MP	3.3+	11.8+	17.1	19.7	20.0-	14.4+
Bagging	16.9+	16.3+	15.5+	15.1	15.7	15.9+
VP	14.3+	14.7+	15.9+	16.4	19.2	16.1+
K*	17.1+	16.9+	17.6	14.8	14.2	16.1+
AdaBoost	13.3+	17.1+	19.3	18.9	17.0	17.1+
PART	19.3	19.8	19.4	18.6	17.2	18.8
J48	22.1	20.9	18.4	17.5	16.1	19.0
SYNTH	21.6	21.5	19.2	17.3	17.0	19.3
TWEBP	22.1	21.5	18.6	17.7	17.5	19.5
WEBP	22.9	21.8	18.8	17.5	16.5	19.5
RF	15.4+	19.4	25.1-	25.5-	20.7-	21.2-
DT	25.2-	23.4	20.5	18.3	18.4	21.2-
JRip	24.4-	22.4	20.9	19.4	20.0-	21.4-
IBk	24.2	21.6	24.1-	25.0-	20.4-	23.1-
REPTree	27.7-	25.3-	23.0-	20.5-	19.6	23.2-
NOP	25.0-	27.1-	27.8-	25.7-	20.7-	25.2-
Ridor	28.1-	27.2-	26.0-	25.1-	24.2-	26.1-
NNge	20.6	24.3	28.0-	29.8-	31.1-	26.8-
OneR	32.1-	30.9-	29.5-	28.4-	26.6-	29.5-
IB1	30.0-	30.6-	30.7-	30.8-	31.3-	30.7-
DS	32.7-	31.7-	30.4-	29.3-	28.2-	30.5-
ZeroR	33.8-	32.8-	32.9-	31.9-	30.7-	32.4-
HP	34.6-	34.2-	34.2-	33.6-	33.4-	34.0-

Table 6.3: Ranks of each pruning algorithm on real world data sets. Obviously, smaller numbers are better. The error percents of the algorithms are shown in parenthesis.

Data sets	NOP	WEBP	TWEBP	SYNTH
aga	2.5(0)	2.5(0)	2.5(0)	2.5(0)
aud	1(24.5)	3(28.4)	4(28.9)	2(27.6)
car	1(5.6)	4(6.3)	2(5.7)	3(6.1)
dna	4(8.4)	1.5(6.1)	1.5(6.1)	3(6.4)
hay	3(28.8)	1.5(27.5)	1.5(27.5)	4(30.0)
krkp	1(0.3)	3(0.5)	2(0.4)	4(0.6)
mon1	2(2.2)	4(2.9)	1(2.0)	3(2.7)
mon2	1(29.3)	2.5(34.3)	2.5(34.3)	4(39.4)
mon3	4(2.7)	2(1.1)	2(1.1)	2(1.1)
nur	1(1.2)	4(3.2)	2(2.4)	3(2.5)
pro	1(25.2)	4(30.3)	2(26.4)	3(27.3)
soy	2(7.9)	1(7.8)	3(8.1)	4(9.8)
tic	4(14.9)	3(14.8)	2(13.9)	1(13.6)
vot	4(6.7)	2(4.8)	1(4.4)	3(5.3)
Avg	2.250(11.3)	2.714(12.0)	2.071(11.5)	2.964(12.3)

J4.8. This shows that the inductive bias of traditional decision tree learners is not suitable for these data sets, and ADATE appears to have had little impact on improving the inductive bias of decision tree learning.

This is further supported by the fact that SYNTH is only slightly better than WEBP and TWEBP.

6.3.2 Real World Data

The real world data sets are the same used in the empirical evaluation of the first synthesized pruning algorithm in section 5.4.2, so please refer to this section for more information about the data sets.

Table 6.3 presents for each data set the rank and error percent of the different pruning algorithms, where the last row contains the average rank and error percent of the algorithms across all the data sets. The error percent is calculated using 10 fold cross-validation and is shown in parenthesis.

SYNTH is the worst pruning algorithm with an average rank of nearly three. WEBP and SYNTH have similar ranks, while the difference between SYNTH and the other two algorithms is much bigger. Still, the differences between the algorithms are too small to reject the null hypothesis of these algorithms being different.

Table 6.4: Average number of nodes in the pruned trees produced by the pruning algorithms on real world data sets.

Data sets	NOP	WEBP	TWEBP	SYNTH	CLEAN
aga	38.0	38.0	38.0	139.0	38.0
aud	136.5	79.4	83.2	690.7	88.5
car	387.6	180.4	221.2	2061.9	218.4
dna	720.2	181.4	172.2	3418.4	222.6
hay	59.6	29.7	30.9	182.5	35.7
krkp	91.4	60.4	66.4	756.6	63.8
mon1	96.7	66.9	77.8	410.3	76.6
mon2	446.4	1.0	1.0	2059.8	191.3
mon3	60.0	17.6	17.6	133.1	17.6
nur	1133.0	501.0	727.2	7346.1	722.0
pro	37.8	19.4	17.0	95.2	21.0
soy	248.1	138.7	143.4	1402.2	131.3
tic	319.6	163.9	216.1	1656.7	180.1
vot	66.4	13.9	4.0	194.1	21.4
Avg	274.4	106.6	129.7	1467.6	144.9

Similarly to the average rank, SYNTH has the highest error percent, WEBP has a somewhat lower error percent, while TWEBP and NOP have distinctively lower error percents.

The amount of pruning applied by the different algorithms is shown in table 6.4. Each row contains the average size of the trees pruned by the algorithms for the different data sets. Since SYNTH adds extra branches that are never traversed, an extra column CLEAN is included that contains the size of the trees produced by SYNTH after the extra branches have been removed. We evaluate SYNTH in terms of this column instead of the original because it better represents the amount of pruning applied.

SYNTH performs more pruning on average than NOP and less pruning than TWEBP and WEBP. However, SYNTH does not systematically produce bigger trees than TWEBP, and it produces smaller trees for several data sets. In this way, SYNTH lies somewhere in between NOP and WEBP in terms of the degree of pruning, where it seems to be positioned more towards WEBP considering it occasionally performs more pruning than TWEBP.

This provides a possible explanation for the poor performance of SYNTH for these data sets because the unpruned trees appear to need either a significant amount of pruning or almost no pruning. Accordingly, SYNTH performs too much pruning for the data sets where little pruning is needed, while it performs too little pruning for the data sets where extensive pruning

is needed.

6.4 Summary

This chapter has explained the second pruning specification, which improves upon the first pruning specification in some areas. The most notable changes made to the first specification were scaling of the arguments given to the `CalculatedDist` constructor, using EBP without tree grafting as initial `f` function and generating more complex data sets with a varying number of attribute and class values.

The synthesized pruning algorithm performed significantly better than the other pruning algorithms across 60000 synthetic data sets, which showed that ADATE was able to improve the initial `f` function. Still, the improvements seemed small and SYNTH was severely outperformed by other classification algorithms on 1000 synthetic data sets. In addition, it performed poorly on real world data sets, although the results were not statistically significant.

These results are similar to the results of the pruning algorithm synthesized for the first specification. Both algorithms performed significantly better than the other pruning algorithms on synthetic data sets and were outperformed by other classification algorithms on the same type of data sets.

Furthermore, none of the synthesized pruning algorithms could be separated statistically from the other pruning algorithms on real world data sets. Despite that, the first pruning algorithm performed the best on these data sets, while the second synthesized pruning algorithm performed the worst.

Chapter 7

Further Work

Solving classification problems through ADATE has been thoroughly investigated in this thesis, yet this investigation is far from complete and there are many more areas that should be further explored. We highlight, in this chapter, the areas that seem the most interesting to investigate for both synthesizing classifiers and improving classification algorithms.

7.1 Synthesizing Classifiers

Little of the expressive power and the flexibility provided by ADATE are used in the synthesis of classifiers in this thesis. As mentioned previously, this was an intentional choice made in order to objectively compare ADATE and the other classification algorithms by avoiding specific customization of the algorithms to the data sets.

In addition, it is hard, if not impossible, to use an expressive representation when the original representation of the data set is more restrictive. This is the case for the data sets used in this paper and for most of the other data sets at UCI machine learning repository, where the attribute-value representation is used.

Nonetheless, it would have been interesting to see what effect any utilization of the more advanced features of ADATE would have on the classifiers synthesized. In the next sections, we highlight one of these features, algebraic data types, and particularly some of its advantages compared to the normal attribute-value representation used in classification. In addition, we describe two related fields of traditional classification, multiple-instance problems and multiple-part problems, where the advanced feature set of ADATE might be better suited.

```
<meals>
    <pizza class="satisfactory">
        <topping>
            <cheese>
                <sort>Mozzarella</sort>
            </cheese>
        </topping>
    </pizza>
    <pizza class="unsatisfactory">
        <topping>
            <Tomatoes/>
        </topping>
    </pizza>
    <tapirsoup class="satisfactory">
        <spiced/>
    </tapirsoup>
</meals>
```

Figure 7.1: An example of structured XML data that cannot be completely represented using the attribute-value encoding. This data is partially taken from [52].

7.1.1 Algebraic Data Types

ADATE-ML supports algebraic data types where a data type consists of one or more of constructors with zero or more arguments of other data types. In this way, it is possible to construct data structures such as lists and trees. However, in the specifications for synthesis of classifiers, the algebraic data types are only used to model the attribute-value representation used in original the data sets so that all the constructors take either no arguments or a single `real`.

The main problem with the attribute-value representation is that it cannot naturally represent structured data. For instance, it is impossible to completely model the XML data in figure 7.1 using this representation. Still, there are workarounds; Unfortunately, these workarounds have negative effects on the representation such as increasing the number of attribute values, introducing attributes only applicable for instances with specific attribute values or introducing redundancy in the attributes so that the same information is represented in different ways [52].

Algebraic data types, on the other hand, are better suited for representing structured data [53, 52]. For instance, figure 7.2 shows how easily the structured data from figure 7.1 can be represented in ADATE-ML using algebraic data types.

```
datatype cheese_sort = Mozzarella | Gorgonzola
datatype topping = Cheese of cheese_sort | Tomatoes | Mushrooms
datatype meal = Pizza of topping | TapirSoup of bool
datatype class = Satisfactory | Unsatisfactory

val (inputs, outputs) = ListPair.unzip[
    (Pizza (Cheese Mozzarella), Satisfactory),
    (Pizza Tomatoes, Unsatisfactory),
    (TapirSoup true, Satisfactory)]
```

Figure 7.2: The data from figure 7.1 represented in ADATE-ML.

A particular attractive feature of algebraic data types is that it allows incremental exposure of attributes. For example, `cheese_sort` from figure 7.2 becomes only available after inspecting `topping` and `meal` through pattern matching. In this way, attributes can be inspected as deeply as necessary, and the attributes are scoped similarly to how ADATE scopes functions, reducing the number of attributes to consider.

7.1.2 Multiple-instance and Multiple-part Problems

Multiple-instance problems (MIP) [54] and multiple-part problems (MPP) [55, 56] are related to traditional classification problems in that the task is to predict the class of an object. However, the object has different properties compared to traditional classification. Instead of being composed of a set of attributes, it is composed of a set of instances.

An example of a multiple-instance problem is the musk problem introduced in [54] where the task is to determine whether a molecule is musky, that is whether it activates the musk molecule. A molecule can have alternative shapes or conformations with different activation properties. In this way, the object is the molecule and the different conformations are the instances.

If at least one of these conformations are musky, the molecule is considered musky. However, the information available for each molecule is incomplete. It is only known whether a molecule is musky, meaning it is unknown which of the specific conformations responsible for the activation.

The instances have different representations in MIP and MPP. In MIP, the instances are alternative representations of the same object, while in MPP, these are different parts of the same object. Considering the musk problem, this would have been a multiple-part problem if each instance corresponded to a separate part of the same molecule, for example an atom.

MIP and MPP do not fit traditional classification algorithms well since

they cannot represent multiple instances. Still, several extensions have been proposed to existing classification algorithms to allow them to solve such problems, yielding competitive results compared to other MIP and MPP methods [56, 57]. Thus, ADATE might be applied with success to these types of problems as well considering the expressive nature and the competitive results with state-of-the-art classification algorithms.

7.2 Improving Classification Algorithms

7.2.1 Other Classification Algorithms

The next natural step in improving classification algorithms is to try to improve some of the other algorithms besides traditional decision tree learning. These could be improved as demonstrated in this thesis by implementing the algorithm in ADATE-ML and targeting different parts of it in synthesis. Naturally, it would be most interesting to attempt to improve state-of-the-art classification algorithms like boosting and SVMs.

For instance, a boosting algorithm like AdaBoost could be improved, either by letting ADATE synthesize the best base classifier to use with AdaBoost for a particular domain, or improving the algorithm as a whole. One problem with AdaBoost is that it can be computationally expensive to execute the base classification algorithm numerous times. Thus, this base algorithm should be fast to execute.

7.2.2 Other Performance Metrics

In this thesis, we considered only one performance metric, the accuracy, when trying to improve decision tree pruning through ADATE. Accuracy is the most popular performance metric, especially for comparing algorithms with each other, and thus it was a natural choice. Nevertheless, there are situations where other performance metrics are better suited than accuracy.

These other performance metrics are often related to predicting the probability of an instance belonging to the different classes instead of predicting a single class. Most classification algorithms can be adapted to return probabilities or more appropriately put return a set of values representing the certainty of each possible prediction. For instance, a decision tree can return probabilities by returning a distribution with the fraction of the instances reaching the leaf for the different classes.

Accurate probability metrics are needed in situations where predictions made by the classifier are processed in some way. For instance, the proba-

bility of each possible prediction might be used by a human expert to decide whether a more thorough and expensive investigation is needed before making the final prediction. In these situations, the squared error and cross-entropy metrics are better suited than accuracy.

In other areas, accurate probabilities are not specifically needed since the probability values are only used to order the instances. For example, in marketing, when sending out direct mail, only a specific portion of the overall population is selected and the goal is to select the portion that maximizes the reply rate. A classifier can be used to find this portion by ordering the instances and selecting the instances (persons) that most likely will reply. For this task, the lift metric is the most appropriate to find the best classifier, while the metrics, break-even point and area under the ROC curve, are better suited in other areas such as information retrieval and medicine. For a more thorough explanation of these metrics please refer [58, 6].

Unfortunately, a classification algorithm that performs well in terms of one metric does not necessarily perform well in terms of another metric. Thus, it would be interesting to investigate whether ADATE can improve classification algorithms in terms of these other metrics. Since this is almost exactly the same problem as improving the accuracy of classification algorithms, only a modification to the output evaluation function of the pruning specifications presented in appendix D and F should be necessary to enable improvement of some other metric than accuracy.

Nevertheless, a comparison of 10 metrics, including the metrics mentioned here, conducted by Caruana and Niculescu-Mizil shows that some metrics are highly correlated [58]. Of the metrics compared, squared error and SAR, a specially designed general purpose metric, were found to select the best classifier on average for the different metrics. Hence, it might be more appropriate to use one of these metrics to improve a classification algorithm on average across all metrics.

Naturally, classifiers could be synthesized in terms of these other metrics as well. ADATE might perform better for these metrics compared to other classification algorithms since ADATE can produce classifiers specifically designed for each metric, while most classification algorithms are directed either at accuracy or squared error.

Chapter 8

Conclusion

This chapter presents our concluding remarks for the two areas of classification explored with the ADATE system in this thesis.

8.1 Synthesizing Classifiers

We have shown how ADATE can be used to induce classifiers for a set classification problems just as easily as other classification algorithms by automatically generating specifications based on the same files utilized by the popular C4.5 decision tree system. In this way, no programming knowledge is needed at all, only knowledge of how to execute programs on the command line.

The execution of these specifications resulted in accurate classifiers and ADATE proved to be competitive with state-of-the-art classification algorithms. It was found, along with Logical Model Trees, to produce on average the most accurate classifiers, although it was only significantly better than 8 of the 32 classification algorithms executed.

Nevertheless, these results were good considering the severe bugs in the synthesis algorithm of the version of ADATE utilized for inducing classifiers, demonstrating the durability and the strength of the systematic population-based search employed by ADATE. However, a new empirical study should be conducted to investigate how ADATE performs without bugs and how much it really affected the ability to find accurate classifiers. For this study, a larger number of data sets should probably be used in order to better distinguish the different algorithms from each other.

Even though we have shown that ADATE can be competitive in terms of accuracy, its computational requirements are large and order of magnitudes higher than the computational requirements of any of the other classification

algorithms used in this thesis. Thus, it is not justifiable in most cases to execute ADATE since simpler methods can be applied with the same result.

Still, there might be problems that are too complex to be sufficiently solved with traditional classification algorithms where the infinite and systematic search utilized by ADATE and the ability to fully represent the problem might be better suited. This seems to be a possible sweet spot for ADATE, and it is probably in this area ADATE should be further explored in terms of synthesis of classifiers, provided that the bugs did not have severe impact on the performance of the synthesized classifiers.

8.2 Improving Classification Algorithms

Improving classification algorithms seems to be a problem that is better suited for ADATE than inducing classifiers since ADATE has been specifically designed for automatic synthesis of algorithms and there is a direct need for algebraic data types, auxiliary functions and recursion.

In this thesis, we have specifically described how one particular type of classification algorithm, namely decision tree learning, can be improved using ADATE by implementing the algorithm in ADATE-ML and targeting different parts of it. This methodology is certainly not restricted to decision tree learning and can be applied to virtually any classification algorithm that is sufficiently light-weight to repeatedly be executed during evaluation.

We targeted decision tree pruning, a specific part of decision tree learning, and showed on two occasions that ADATE was able to improve the pruning algorithm chosen as the start program with statistical significance for a fairly general domain. Both of the synthesized pruning algorithms outperformed the other pruning algorithms when evaluated on synthetic data sets selected according to a probability distribution that attempts to model the effects of not having enough information available in real world data sets.

In addition, one of the two synthesized pruning algorithms performed better than the other pruning algorithms across real world data sets, although the differences between the algorithms were not statistically significant according to the Friedman test.

Still, the improvements to decision tree learning contributed by these pruning algorithms seemed rather small considering their bad performance on synthetic data sets compared to numerous other classification algorithms. The first pruning algorithm was outperformed by 15 algorithms and the second pruning algorithm was outperformed by 17. However, only one and two of these algorithms were traditional decision tree learners or used traditional decision tree learners in some way, suggesting that the inductive bias of tradi-

tional decision tree learners is not suited for these data sets and that ADATE is not able to extensively modify the inductive bias.

Even though the improvements to decision tree learning seemed rather small, we have at least shown that ADATE made some improvements, both in general and for a particular domain, to the initial pruning algorithm, and that it might be possible to automatically improve other classification algorithms as well. In addition, advances in the ADATE system and in computational power of computers might in the future lead to improvements of state-of-the-art machine learning algorithms like boosting and SVM.

References

[1] J. Chilo, R. Olsson, S.-E. Hansen, and T. Lindblad, "Classification of infrasound events with various machine learning techniques," in *Proceeding of CCCT'07*, 2007.

[2] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Mach. Learn.*, vol. 6, no. 1, pp. 37–66, 1991.

[3] J. G. Cleary and L. E. Trigg, "K*: an instance-based learner using an entropic distance measure," in *Proc. 12th International Conference on Machine Learning*. Morgan Kaufmann, 1995, pp. 108–114.

[4] C. Atkeson, A. Moore, and S. Schaal, "Locally weighted learning," *AI Review*, vol. 11, pp. 11–73, April 1997.

[5] E. Frank, M. Hall, and B. Pfahringer, "Locally weighted naive bayes," in *Proceedings of the 19th Annual Conference on Uncertainty in Artificial Intelligence (UAI-03)*. San Francisco, CA: Morgan Kaufmann, 2003, pp. 249–25.

[6] E. F. Ian H. Witten, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[7] S. le Cessie and J. van Houwelingen, "Ridge estimators in logistic regression," *Applied Statistics*, vol. 41, no. 1, pp. 191–201, 1992.

[8] N. Landwehr, M. Hall, and E. Frank, "Logistic model trees," *Mach. Learn.*, vol. 59, no. 1-2, pp. 161–205, 2005.

[9] J. Friedman, T. Hastie, and R. Tibshirani, "Additive logistic regression: a statistical view of boosting," 1998.

[10] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of 5-th Berkeley Symposium on*

Mathematical Statistics and Probability, vol. 1. Berkeley, University of California Press, 1967, pp. 281–297.

[11] J. C. Platt, "Fast training of support vector machines using sequential minimal optimization," *Advances in kernel methods: support vector learning*, pp. 185–208, 1999.

[12] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to platt's smo algorithm for svm classifier design," *Neural Comput.*, vol. 13, no. 3, pp. 637–649, 2001.

[13] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," in *Computational Learing Theory*, 1998, pp. 209–217.

[14] F. Rosenblatt, "The perceptron: A probalistic model for information storage and orginazation in the brain," *Psychological Review*, vol. 65, pp. 386–407, 1958, (Reprinted in Neuro-computing (MIT Press, 1988)).

[15] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.

[16] ——, *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[17] R. Kohavi, "Scaling up the accuracy of Naive-Bayes classifiers: a decision-tree hybrid," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 202–207.

[18] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

[19] E. Frank, Y. Wang, S. Inglis, G. Holmes, and I. H. Witten, "Using model trees for classification," *Mach. Learn.*, vol. 32, no. 1, pp. 63–76, 1998.

[20] J. R. Quinlan, "Learning with Continuous Classes," in *5th Australian Joint Conference on Artificial Intelligence*, 1992, pp. 343–348.

[21] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[22] Y. Freund and L. Mason, "The alternating decision tree learning algorithm,," in *Proc. 16th International Conf. on Machine Learning*. Morgan Kaufmann, San Francisco, CA, 1999, pp. 124–133.

[23] W. W. Cohen, "Fast effective rule induction," in *Proc. of the 12th International Conference on Machine Learning*, A. Prieditis and S. Russell, Eds. Tahoe City, CA: Morgan Kaufmann, July 9–12, 1995, pp. 115–123.

[24] X. Xu and E. Frank, "Jrip," javadoc. [Online]. Available: http://weka.sourceforge.net/doc/weka/classifiers/rules/JRip.html

[25] R. Kohavi, "The power of decision tables," in *ECML '95: Proceedings of the 8th European Conference on Machine Learning*. London, UK: Springer-Verlag, 1995, pp. 174–189.

[26] E. Frank and I. H. Witten, "Generating accurate rule sets without global optimization," in *ICML '98: Proceedings of the Fifteenth International Conference on Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 144–151.

[27] G. Holmes, M. Hall, and E. Frank, "Generating rule sets from model trees," in *AI '99: Proceedings of the 12th Australian Joint Conference on Artificial Intelligence*. London, UK: Springer-Verlag, 1999, pp. 1–12.

[28] B. Martin, "Instance-based learning : Nearest neighbor with generalization," Master's thesis, University of Waikato, Hamilton, New Zealand, 1995.

[29] S. Roy, "Nearest neighbor with generalization," 2002, unpublished.

[30] G. Demiroz and H. A. Guvenir, "Classification by voting feature intervals," in *European Conference on Machine Learning*, 1997, pp. 85–92.

[31] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[32] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm," in *International Conference on Machine Learning*, 1996, pp. 148–156. [Online]. Available: citeseer.ist.psu.edu/freund96experiments.html

[33] J. R. Olsson, "Inductive functional programming using incremental program transformation and execution of logic programs by iterative-deepening a* sld-tree search," Ph.D. dissertation, University of Oslo, October 1994.

[34] ——, "Inductive functional programming using incremental program transformation," *Artificial Intelligence*, vol. 74, no. 1, pp. 55–8, 1995.

[35] ——, "Population management for automatic design of algorithms through evolution," in *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*. Anchorage, Alaska, USA: IEEE Press, 5-9 1998, pp. 592–597.

[36] J. H. Holland, *Adaptation in natural and artificial systems*. Cambridge, MA, USA: MIT Press, 1992.

[37] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, december 1992.

[38] R. E. Korf, "Depth-first iterative-deepening: an optimal admissible tree search," *Artif. Intell.*, vol. 27, no. 1, pp. 97–109, 1985.

[39] G. Vattekar, *ADATE User Manual*, march 2006.

[40] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," in *European Conference on Computational Learning Theory*, 1995, pp. 23–37.

[41] C. B. D.J. Newman, S. Hettich and C. Merz, "UCI repository of machine learning databases," 1998. [Online]. Available: http://www.ics.uci.edu/∼mlearn/MLRepository.html

[42] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the American Statistical Association*, vol. 32, no. 200, pp. 675–701, 1937.

[43] ——, "A comparison of alternative tests of significance for the problem of m rankings," *The Annals of Mathematical Statistics*, vol. 11, no. 1, pp. 86–92, 1940.

[44] R. L. Iman and J. M. Davenport, "Approximations of the critical region of the friedman statistic," *Communications in Statistics*, pp. 571–595, 1980.

[45] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 1, pp. 65–70, 1979.

[46] D. H. Wolpert, "The supervised learning no-free-lunch theorems," in *6th Online World Conference on Soft Computing in industrial applications*, 2001.

[47] R. E. Schapire, "The strength of weak learnability," *Mach. Learn.*, vol. 5, no. 2, pp. 197–227, 1990.

[48] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ, 2003.

[49] J. Mingers, "An empirical comparison of pruning methods for decision tree induction," *Mach. Learn.*, vol. 4, no. 2, pp. 227–243, 1989.

[50] F. Esposito, D. Malerba, and G. Semeraro, "A comparative analysis of methods for pruning decision trees," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 19, no. 5, pp. 476–491, 1997.

[51] J. R. Quinlan, "Simplifying decision trees," *Int. J. Man-Mach. Stud.*, vol. 27, no. 3, pp. 221–234, 1987.

[52] M. Mottl, "Using algebraic datatypes as uniform representation for structured data," *Submitted to: Machine Learning Journal, Special Issue on Inductive Logic Programming and Relational Learning*, 2003.

[53] ——, "Modelling large datasets using algebraic datatypes: A case study of the confman database," Austrian Research Institute for Artificial Intelligence, Tech. Rep. 2002-27, May 2002.

[54] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Perez, "Solving the multiple-instance problem with axis-parallel rectangles," *Artificial Intelligence*, vol. 89, no. 1-2, pp. 31–71, 1997.

[55] J.-D. Zucker and J.-G. Ganascia, "Selective reformulation of examples in concept learning," in *International Conference on Machine Learning*, 1994, pp. 352–360.

[56] J.-D. Zucker and Y. Chevaleyre, "Solving multiple-instance and multiple-part learning problems with decision trees and decision rules. application to the mutagenesis problem," in *Internal Report, University of Paris 6*, 2000.

[57] J. Wang and J.-D. Zucker, "Solving the multiple-instance problem: A lazy learning approach," in *Proc. 17th International Conf. on Machine Learning*. Morgan Kaufmann, San Francisco, CA, 2000, pp. 1119–1125.

[58] R. Caruana and A. Niculescu-Mizil, "Data mining in metric space: An empirical analysis of supervised learning performance criteria," in *Proceedings of the Tenth International Conference on Knowledge Discovery and Data Mining (KDD'04)*, 2004. [Online]. Available: citeseer.ist.psu.edu/caruana04data.html

List of Figures

List of Tables

Appendix A

Infrasound Article

Classification of Infrasound Events with Various Machine Learning Techniques

José CHILO
Royal Institute of Technology, S-106 91 Stockholm
and University of Gävle, S-801 76 Gävle, Sweden
Roland OLSSON
Ostfold University College, N-1757 Halden, Norway
Stig-Erland HANSEN
Ostfold University College, N-1757 Halden, Norway
and
Thomas LINDBLAD
Royal Institute of Technology, S-106 91 Stockholm, Sweden

ABSTRACT

This paper presents classification results for infrasonic events using practically all well-known machine learning algorithms together with wavelet transforms for pre-processing. We show that there are great differences between different groups of classification algorithms and that nearest neighbor classifiers are superior to all others for accurate classification of infrasonic events.

Keywords: Classification, Machine Learning, Pattern Recognition, Wavelets

1. INTRODUCTION

Infrasound is low frequency sound, typically of a frequency of a few Hertz to 20 Hertz. Due to its inherent properties, infrasound can travel distances of many hundreds of kilometers. Infrasound signals can result from nuclear explosions, volcanic eruptions, mountain associated waves, auroral waves, earthquakes, meteors, avalanches, severe weather, quarry blasting, air/spacecraft, gravity waves, microbaroms, opening and closing of doors, trains and helicopters to name but a few. An infrasound monitoring system operating locally like the Swedish-Finnish Infrasound Network[1] or worldwide like CTBTO[2] must be capable of detecting and verifying infrasonic signals of interest and discriminating them from other unwanted infrasonic signals. Characterizing, discriminating and classifying infrasonic events therefore are tasks with possibly far reaching applications in very different disciplines.

An important element for successful classification of infrasound data is the pre-processing techniques used to form a set of feature vectors that can be used to train and test the classifiers. In this work we use continuous wavelet transforms to pre-process infrasound data. Wavelet transformations have proven to be a valuable tool for signal characterization [1,2]. The wavelet transform methods developed over the years at IRF Umeå [3, 4] are used in this paper.

[1] http://www.umea.irf.se/maps/
[2] http://www.ctbto.org/

Machine learning provides the technical basis to extract implicit, previously unknown, and potentially useful information from infrasound data. The idea is to build computer programs that sift through infrasound datasets automatically, seeking regularities or patterns. Strong patterns, if found, will likely generalize to make accurate classifications on new data. We use a variety of machine learning methods, including neural nets, support vector machines, decision trees, association rules, linear models, Bayes nets and others.

The advantages of neural network based approaches for classifying infrasonic events have been recognized for a while [5, 6]. Neural networks are considered to be powerful classification tools because of their non-linear properties and the fact that they make no explicit assumptions about the distribution of the data. We experimented with several neural network classifiers, including back-propagation classifiers, minimum least square linear classifiers, normal densities based quadratic classifiers, automatic neural network classifiers and random neural network classifiers. Comparing their performance, we reached the conclusion that neural network classifiers by back-propagation work the best among neural net techniques in our case, but are clearly inferior to many other machine learning methods.

2. FEATURE SELECTION

Wavelet transforms methods are used to pre-process infrasound data. The data pre-processing steps to extract feature vectors are as follows.

1.For a time series of N values a Morlet wavelet transform is performed with 128 dilations. Thus, three matrices, **A**, **R** and **I**, are obtained. The matrix **A** is a matrix of magnitudes of wavelet coefficients, w_{ij}:

$$A = \{| w_{ij} |\} \quad i = 1,...N \quad j = 1,...128 \qquad (1)$$

R and **I** contain the real and imaginary parts of w_{ij}.

2.A kind of band-pass filtering of wavelet coefficient magnitudes is performed. The entire range of coefficient magnitudes, 0 to max (w_{max}), is divided into 20 intervals

such that the *k-th* interval is limited by:

$$w_{\max} * \frac{(k-1)}{20} \text{ and } w_{\max} * \frac{k}{20} \; ; k = 1,...20 \qquad (2)$$

For each *k* the coefficients outside the range defined by Eq. (2) are identified and zeroed in matrices **R** and **I**, creating two new matrices $\mathbf{R_k}$ and $\mathbf{I_k}$. The inverse wavelet transform is performed using $\mathbf{R_k}$ and $\mathbf{I_k}$ and a new version of the original time series, $y_k(t_i)$ is created. Thus, the time series $y_k(t_i)$ is what the signal would look like if only a narrow range of spectral densities would be present in the signal.

3.The operation is repeated 20 times over the range of coefficient magnitudes. A new real-valued matrix **Z**, consisting of 20 rows and N columns is created. Each row corresponds to a time series, $y_k(t_i)$.

4.Each row of the matrix **Z** is wavelet transformed as in step 1, resulting in 20 matrices. Then these matrices are time-averaged (average along rows) leading to 20 arrays with 128 elements. A new matrix **Y** is constructed with the 20 arrays as rows. This matrix **Y** is what we call Time Scale Spectrum (TSS) of the time series.

A 3-D plot of the matrix **Y** may be constructed, showing the time scale (1/frequency) of the signal on the x-axis, the wavelet coefficient magnitude of the original signal, in percent of its max value, on the y-axis and the wavelet coefficient magnitude (power spectral density) of the decomposed components as the colour scale.

For the infrasound signals the TSS may be useful to resolve different frequency components. This feature extraction process is invariant with respect to record length, sampling frequency, signal amplitude and time sequence length. Figure 1 shows an infrasound signal train with 2^{12} = 4096 components, representing 227.55 seconds sampled at 18 Hz and its TSS.

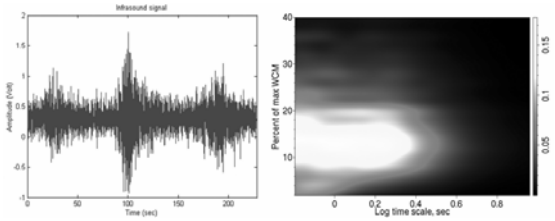

Fig. 1. *The infrasound signal from a meteorite and its TSS*

3. CLASSIFIERS

In this section, we first describe classical neural nets trained with back-propagation and then machine learning algorithms in WEKA.

Back-propagation neural nets (BPNNs) typify supervised learning, where the task is to learn to map input vectors to desired output vectors. The back-propagation learning algorithm modifies feed-forward connections between the input and the hidden units, and the hidden and outputs units, so that when an input vector is presented to the input layer, the output layer's response should be the desired output vector. During training, the error caused by the difference between the desired output vector and the output layer's response to an input vector propagates back through connections between layers and adjusts appropriate connection weights so as to minimize the error [7].

WEKA [8] contains practically all common machine learning algorithms except neural nets. However, not all of those algorithms have support for both numerical features and nominal classes. In addition, we experienced problems with four of the algorithms, making it impossible to use them. All in all, we ended up running 22 different algorithms with their default parameters if nothing else is stated.

These algorithms are grouped into five groups in WEKA according to what models they create. The first group, Bayes, includes algorithms where learning results in Bayesian models**.** NaiveBayes is an implementation of the standard naïve Bayes algorithm, where a normal distribution is used for numerical features. BayesNet creates a Bayesian Network with the ability to represent the same model as NaiveBayes or other more complex models where the independence between features is not assumed.

The second group, Lazy, is comprised of algorithms that delay construction of classifiers until classification time. IB1 is a nearest-neighbor algorithm classifying an instance according to the nearest neighbor identified by the Euclidean distance as explained in [9]. IBK is similar to IB1 except that the k nearest neighbors are used instead of only one. We determined the appropriate number of neighbors using leave-one-out cross-validation. Another algorithm is LWL (Locally weighted learning), which differs from the other two algorithms since it only uses a nearest-neighbor algorithm to weight the instances in the training set before applying another classification algorithm to them. We chose naïve Bayes because it is recommended for classification problems by the creators of WEKA.

The third group, Rules, contains methods that create classification rules. OneR is the simplest of all the rule inducers and learns a single rule using only a single feature. The other four algorithms are more complex since they create several rules. NNge is a nearest-neighbor algorithm which learns rules based on the hyper rectangles it divides the instance space into [10]. JRip is an implementation Cohen's RIPPER [11]. RIPPER creates first a default rule and then recursively develops exceptions to it. Part constructs rules based on partial decision trees.

The fourth group, Functions, contains algorithms representing their learnt models as mathematical formulas.. SMO is a sequential optimization algorithm for building Support Vector Machines (SVMs) [12]. We used a polynomial kernel which is the default in WEKA. RBFNetwork is an implementation of radial basis functions, and SimpleLogistic constructs linear logistic regression models.

The fifth group, Trees, includes algorithms that create trees as models. Four of the six tree inducers create trees with a single class at the leaves. RandomTree learns a multi-level tree constructed by randomly choosing the splitting criterion. RandomForest is an implementation of Breiman's random forest [13], where bagging and random trees are combined. J48 is an implementation of the popular C4.5 [14]. REPTree is similar to C4.5 since it finds the splitting criteria based on information gain, but it uses reduced-error-pruning to prune the tree instead of pessimistic training error.

The last two algorithms have models in the leaves instead of a specific class. NBTree builds a tree with naïve Bayes classifiers at the leaves, where reduced-error-pruning controls the depth of the tree. LMT creates a tree with linear logistic regression models at the leaves.

The last group, Miscellaneous, contains algorithms that do not fit into any of the other groups. HyperPipes finds ranges (max and min values for numerical features) for each feature and class pair. An instance is classified as the class with the most "hits" into its ranges. VFI, on the other hand, finds intervals for each feature, and attributes each class according to number of instances with the class in the training set for the specific interval. Voting is used to select the final class for an instance. Both of these algorithms are simple compared to the other algorithms and extremely fast.

4. EXPERIMENTAL RESULTS

The feature vectors, TSS, were extracted from time-domain event signals resulting in two dimensional 20x128 matrices. These matrices were converted to 2560-element one dimensional feature vectors. Figure 2 shows two sets of feature vectors of the two different types of events.

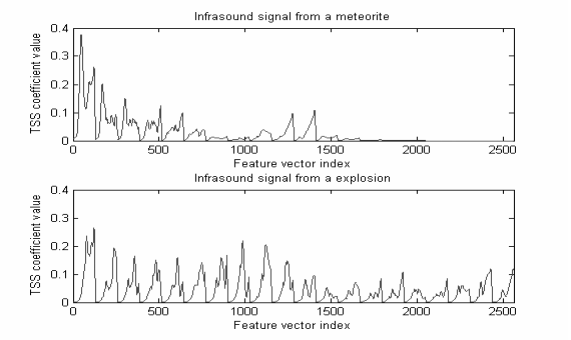

Fig. 2. *Two feature vectors*

Experiment 1: One Infrasound Category

In this experiment, we made a total of 200 infrasound measurements of 10 different doors being opened and closed. We chose to use 100 of these examples for training and the remaining 100 for testing.

The experiment was conducted with the MatLab neural network toolbox for BPNN and the WEKA toolbox for other machine learning algorithms. Each classifier was trained using 10 samples of every door and tested with a different set of 10 samples of every door. The classification results are shown in table 1 for BPNN and in table 2 for the WEKA machine learning algorithms.

The 2560/200/10 architecture was selected for BPNN, which means that the input layer has 2560 neurons, the hidden layer 200 neurons (this number was picked after we experimented with different number of neurons, see Table 1) and the output layer 10 neurons. The output layer is to produce target output as [{1000000000},{0100000000},…}]. The network is a two-layer log-sigmoid/log-sigmoid network. The log-sigmoid transfer function was picked because its output range (0 to 1) is perfect for learning to output Boolean values. All training was done using back-propagation with both adaptive learning rate and momentum. The network was trained for a maximum of 5000 epochs or until the network mean squared error (MSE) falls beneath 0.01. The final MSE was 0.00982 after 300 training epochs. The results give a 24 % error.

Table 1. *Results from BPNN*

Architecture	Training epochs	Error %
2560/20/10	903	34
2560/40/10	1281	31
2560/100/10	307	28
2560/150/10	300	27
2560/200/10	300	24
2560/300/10	315	23
2560/400/10	290	22

The classification results for machine learning methods in WEKA vary greatly between algorithms and between different groups of algorithms, see Table 2.

Table 2. *Results from WEKA*

Group	Algorithm	Error %
Bayes	NaiveBayes	12
	BayesNet	12
Lazy	IB1	7
	IBK (Cross-validation)	7
	LWL (NaïveBayes)	11
	IBK (5)	12
Rules	NNge	10
	Part	27
	Ridor	34
	JRip	40
	DecisionTable	41
	OneR	60
Functions	SMO	8
	RBFNetwork	13
	SimpleLogistic	15
Trees	RandomForest	14
	LMT	15
	J48	23
	REPTree	33
	RandomTree	37
Misc	Hyperpipes	11
	VFI	42

Bayes is one of the better groups. BayesNet have similar performance to NaiveBayes, possibly due to construction of a network similar to NaiveBayes.

Lazy is by far the best group of algorithms. The best algorithms in this group are IB1 and IBK with seven percent error on the test data. In addition, the equal result of IB1 and IBK suggest that a single neighbor was chosen during cross-validation. To investigate this further, we tested running IBK with five neighbors which resulted in an error percent of 12.

LWL achieved poorer results than the other two algorithms in the group. This might be a result of it not using the nearest neighbor algorithm directly, but only for weighting the training set. Interestingly, it has similar performance to NaiveBayes, which is the algorithm used as base learner for LWL. Thus, it appears that local weighting has a small or no impact for this dataset.
Functions have done well as a group, which is not particularly surprising based on the fact that the algorithms in this group tend to handle numerical features well. SMO is one of the best algorithms overall, which shows that support vector machines are worth trying for signal classification. It seems that the more complex the method, the better the result.

The Trees group is split into two groups according to the results. In the first group are RandomForest and LMT. These are methods that either build trees with models at their leaves or combine several trees. The other tree inducers, which create trees with a single class at the leaves, achieve much poorer results. These differences in performance might be the result of more complex trees having to be induced by the simpler algorithms due to simpler leaves, numerous classes and only numerical features resulting in binary splitting. In addition, the models of LMT and NBTree, logistic regression and naïve Bayes, perform well separately for this dataset.
The Rules group is probably the worst group of all. NNge is the only rule inducer that performs well, and it is a nearest neighbor algorithm. The other algorithms perform poorly, possibly due to being better at handling nominal rather than numerical features.

The algorithms in the Miscellaneous group vary greatly in performance. Hyperpipes have a low error percent, while VF1 have a high error percent. This shows that one should always test the simplest algorithms first before using the more complex and computational intensive methods.

Experiment 2: Four Categories

Four categories of infrasound events are of interest in this section. The data were collected from different infrasound sensor arrays with different geometries and different locations. Details of the data are given in Table 3.

Table 3. *Infrasound data summary*

Event Type	No. of Events	No. of Samples
Meteorites	4	30
Vehicle	3	27
Man-made explosion	8	24
Opening-closing doors	10	27

To measure the classification accuracy, a 10-fold cross-validation technique is used in this experiment. That is, the whole dataset is partitioned into 10 subsets. Then 9 of the subsets are used as the training set, and the tenth is used as the test set. This process is repeated 10 times, once for each subset used as the training set. Classification performance comes from the average of these 10 runs. This technique ensures that the training and test sets are disjoint, see Table 4.

Table 4. *Results from WEKA*

Group	Algorithm	Error %
Bayes	NaiveBayes	16
	BayesNet	6
Lazy	IB1	1
	IBK (Cross-validation)	1
	LWL (NaïveBayes)	12
	IBK (5)	6
Rules	NNge	19
	Part	12
	Ridor	13
	JRip	19
	DecisionTable	10
	OneR	21
Functions	SMO	6
	RBFNetwork	12
	SimpleLogistic	6
Trees	RandomForest	7
	LMT	5
	J48	11
	REPTree	19
	RandomTree	18
Misc	Hyperpipes	20
	VFI	6

5. CONCLUSIONS

Features based on wavelet transform methods proven effective for the analysis and characterization of infrasound signals when combined with the best state-of-the-art machine learning methods from the WEKA toolbox. The best of these methods, IB1, yields only seven percent error on the one category test data whereas neural networks as implemented in MatLab gave more than a twenty percent error for all the architectures that we tried.
This shows how important it is to choose the appropriate machine learning algorithm for a given problem domain and that there are huge and domain specific variations between the algorithms.

ACKNOWLEDGMENT

The authors would like to thank Dr. Clark S. Lindsey for valuable comments on the manuscript.

References

[1] Liszka, L. “Categorization of Infrasonic Sources”. Paper presented at CTBT Infrasound Workshop 2000, Passau, Germany.

[2] Schmitter, E. D. “Characterisation and Classification of Natural Transients”, *Transactions on Engineering, Computing and Technology*, vol. 13, May 2006.

[3] Liszka L. and Holmström M., “Extraction of a deterministic component from ROSAT X-ray data using a wavelet transform and the principal component analysis”. *Astron. Astrophys. Suppl. Ser.* , 140, 125-134, November 1999.

[4] Liszka L., *Cognitive Information Processing in Space Physics and Astrophysics*, Pachart Publishing House, Tucson, Arizona, 2003.

[5] Ham F. and Park S., “A Robust Neural Network Classifier for Infrasound Events Using Multiple Array data”, *IEEE International Joint Conference NN*, vol. 3, 2615-2619, May 2002.

[6] Ham F., Rekab K., Park S., Acharyya R. and Lee Y., “Classification of infrasound Events Using Radial Basis Function Neural Network”. *IEEE International Joint Conference NN*, vol. 4, 2649-2654, July 2005.

[7] Wang D., “Pattern Recognition: Neural Networks in Perspective”. Ohio State University 1993.

[8] Witten I. H. and Frank E., *Data mining: Practical Machine Learning Tools and Techniques*, 2nd Edition, Morgan Kaufmann Publishers, San Mateo, CA, 2005.

[9] Aha D.W., “Tolerating noisy, irrelevant, and novel attributes in instance-based learning algorithms”, *International Journal of Man-Machine Studies*, vol. 36(2), 267-287, 1992.

[10] Martin B., “Instance-Based learning: Nearest Neighbor With Generalization”, *Master Thesis*, University of Waikato, Hamilton, New Zealand, 1995.

[11] Cohen W. W., "Fast Effective Rule Induction", *Proceedings of the 12th International Conference, on Machine Learning*, 115-123, 1995.

[12] Platt J., “Sequential minimal optimization: A fast algorithm for training support vector machines”, *Technical Report 98-14*, Microsoft Research, Redmond, Washington, April 1998.

[13] Breiman L., ”Random Forests”, *Machine Learning*, vol. 45(1), 5-32, 2001.

[14] Quinlan R., *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.

Appendix B

ADATE Boost

```
1  (* start boosting *)
2  exception AssertionError of string;
3
4  fun assert(false) = raise AssertionError("")
5    | assert(true) = ();
6
7  (*
8   * all instances start with a weight so that the sum of all weights is one
9   * @param numInstances the number of instances to make weights for
10  *)
11 fun createInitialWeights(numInstances) =
12     Vector.tabulate(numInstances, fn _ => 1.0 / real(numInstances));
13
14 (*
15  * Returns the error rate based on the results and the corresponding weights
16  * More specifically, the errorRate is the sum of all weights for the
17  * incorrectly classified instances
18  * @param weights the weights of the different instances
19  * @param results a boolean vector which specify whether an instance was
20  * correctly classified or not
21  *
22  * @return the errorRate
23  *)
24 fun errorRate(weights, results) =
25     Vector.foldri (fn(i, weight, errorRate) => if Vector.sub(results, i) then errorRate
26         else errorRate + weight) 0.0 weights
27
28 (*
29  * Reweights the weights so that instances that are classified correctly are
30  * divided by e * (1 - e), where e is the weighted error
31  * @param results a boolean vector with the results of the bestIndividiual
32  * @param weights a real vector with the weight of each instance
33  * @param errorRate the errorRate which be used to reweight
34  * @return the reweighted vector
35  *
36  *)
37 fun reweight(Weights, Results, ErrorRate) =
38 let
39     val () = assert(Vector.length(Weights) = Vector.length(Results));
40
41     val Beta = ErrorRate / (1.0 - ErrorRate);
42
43     fun updatedWeight(Index, Weight) =
44         if Vector.sub(Results, Index) then Weight * Beta else Weight;
45
46 in
47      Vector.mapi updatedWeight Weights
48 end;
49
50 (*
51  * Normalizes the weights so that the sum of all weights are one
52  * @param weights the weights which will be normalized
53  * @return normalized vector
54  *)
55 fun normalize weights = let
56     val sumWeights = Vector.foldr (op +) 0.0 weights;
57 in
```

```
58      Vector.map (fn (weight) => weight / sumWeights) weights
59  end;
60
61
62  (*
63   * returns the output with the most votes.
64   * If there are no votes, the function returns NONE
65   * @param eq function which returns whether two output are the same
66   * @param weightedOutputs a list of the outputs of the boosted individuals and
67   * their weights (the errors in training)
68   * @return the output of the voting. NONE is returned if none of the individuals
69   * vote
70   *)
71  fun majorityVote(eq, []) = NONE
72    | majorityVote(eq: 'output * 'output -> bool,
73      weightedOutputs: (real * 'output)list): 'output option = let
74
75      (*
76       * adds the weight of x and y if they are equal otherwise y is returned
77       *)
78      fun addWeightedOutput(
79              (wx,cx), y as (wy, cy)) =
80          if eq(cx, cy) then (wy  -  Real.Math.ln(wx / (1.0 - wx)), cy) else y;
81
82      (*
83       * returns the sum of all weights of all weightClasses in l that
84       * have an class equal to class
85       *)
86      fun sumWeightedOutput l ((weight, output): real * 'output) =
87          List.foldr addWeightedOutput (0.0, output) l;
88
89      (*
90       * returns a list where the sum of each item's class is summed
91       *)
92      fun sumWeightedOutputs l = List.map (sumWeightedOutput l) l;
93
94      (*
95       * returns true if the weight of x is bigger than y's, otherwise false
96       *)
97      fun greater (x as (wx, cx): real * 'output, y as (wy, cy): real * 'output) =
98          if wx > wy then x else y;
99
100 in
101     SOME (#2( List.foldr greater (hd weightedOutputs) (sumWeightedOutputs
            weightedOutputs) ))
102 end
103
104
105
106 (*
107  * Selects the indidual with the lowest error rate according to the specified
108  * weights and results
109  *
110  * @param Weights the weights of each instance
111  * @param ResultsForIndis a vector with the individuals and their results for
112  * each instance
113  *
114  * @return the index of the individual with the lowest error rate along with its
115  * error rate
116  *)
117 fun selectBestIndiI(
118     Weights: real vector,
119     ResultsForIndis: ('a * bool vector) vector): (int * real) =
120 let
121     val ErrorRates = Vector.map (fn (Indi, Results) => errorRate(Weights, Results))
            ResultsForIndis;
122
123     fun lessErrorRate(I,
124         ErrorRateX,
125         Y as (IY, ErrorRateY)) =
126         if ErrorRateX < ErrorRateY then (I, ErrorRateX) else Y;
127 in
128     Vector.foldri lessErrorRate (0, Vector.sub(ErrorRates, 0)) ErrorRates
129 end
130
131
132 (*
133  * Performs boosting by selecting a program at each iteration that is best on
134  * the weighted dataset. The dataset is reweighted after each iteration
135  * according to the performance of the selected individual, so that instances
136  * that it got wrong receive higher weights.
137  * @param numIterations the number of iterations to use
138  * @param IndiList the list of individual to select from
```

```
139 |  *
140 |  * It relies on the isCorrect function which returns whether an element in
141 |  * OutputEvals is true or not
142 |  *
143 |  * @return a list of programs along with their weights
144 |  *)
145 | fun boosting(
146 |     NumIterations: int,
147 |     IndiList: individual list): (real * individual) list =
148 | let
149 |
150 |     val Indis = Vector.fromList IndiList;
151 |     val NumIndis = Vector.length Indis;
152 |
153 |     (*
154 |      * Change the number of instances if not enough indis
155 |      * Should perhaps raise an exception instead
156 |      *)
157 |     val NumIterations = if NumIterations > NumIndis then NumIndis else NumIterations;
158 |
159 |     (*
160 |      * Creates a new vector where item i is removed
161 |      * @param Indis the orginal Vector
162 |      * @param I the index of the item to remove
163 |      * @return a vector where item i is removed
164 |      *)
165 |     fun removeAt(Indis, I) =
166 |         if I < 0 orelse I >= (Vector.length Indis) then
167 |             raise Subscript
168 |         else
169 |             VectorSlice.concat [VectorSlice.slice (Indis, 0, SOME I), VectorSlice.slice
                      (Indis, I + 1, NONE)];
170 |
171 |     (*
172 |      * a bool Vector Vector where one item holds whether an Indi is correct on
173 |      * an given instance
174 |      *)
175 |     val Results  =
176 |         Vector.map
177 |             (fn Indi =>
178 |                 (Indi, Vector.map
179 |                     (fn OutputEval => isCorrect(OutputEval))
180 |                     (outputEvals Indi)))
181 |             Indis;
182 |
183 |
184 |     fun boostingHelper(Iteration, Weights, IndiAndResults) =
185 |         if Iteration >= NumIterations then
186 |             nil
187 |         else
188 |             let
189 |                 (*
190 |                  * Select the best indi
191 |                  *)
192 |                 val (BestIndiI, ErrorRate) = selectBestIndiI(Weights, IndiAndResults );
193 |
194 |                 val (BestIndi, BestIndiResults) = Vector.sub(IndiAndResults, BestIndiI);
195 |
196 |             in
197 |
198 |                 (*
199 |                  * Requirement of boosting that the classifier is better than
200 |                  * 0.5, cannot use error = 0.0 since division will be inf when
201 |                  * voting
202 |                  *)
203 |                 if ErrorRate > 0.5 orelse ErrorRate <= 0.0 then
204 |                     (*
205 |                      * only include if first iteration since it will overpower
206 |                      * the other members in a committee larger than one
207 |                      *)
208 |                     if Iteration = 0 then
209 |                         [(ErrorRate, BestIndi)]
210 |                     else
211 |                         nil
212 |                 else
213 |                     (*
214 |                      * calls boosting helper with normlized, reweighted
215 |                      * weights and the best Indi removed from the Indis list
216 |                      *)
217 |                     (ErrorRate, BestIndi) :: boostingHelper(
218 |                         Iteration + 1,
219 |                         normalize(
220 |                             reweight(
```

```
221                                     Weights, BestIndiResults, ErrorRate)),
222                             removeAt(IndiAndResults, BestIndiI))
223             end;
224 in
225     boostingHelper(0, createInitialWeights(NumberOfInputs), Results)
226 end;
227
228
229 (*
230  * returns a list with the output for each instance after each individual has
231  * voted. Instances where no votes are received are removed.
232  *
233  * @param eq equality function for whether to outputs are equal
234  * @param OutputForIndis a list of vectors with the outputs and weights of
235  * the different individuals
236  * @return a list of tupples, where each tupple consists of an instance number
237  * and the output after voting for that instances. However, only instances that
238  * that receive any votes are returned
239  *
240  *)
241 fun weightedTestOutputs(eq, []) = []
242   | weightedTestOutputs(
243     eq: 'output * 'output -> bool,
244     OutputsForIndis: ((real * 'output option) vector) list): (int * 'output) list =
245 let
246     (* number of outputs for test instances*)
247     val NumIndis = Vector.length(hd OutputsForIndis)
248
249     (*
250      * removes all NONE from a list
251      *)
252     fun removeNones( nil ) = nil
253       | removeNones((_, NONE) :: xs ) = removeNones(xs)
254       | removeNones((Weight, SOME(X)) :: xs ) = (Weight, X) :: removeNones(xs);
255
256
257     val Lower = NumberOfInputs
258
259     (*
260      * creates a list where each item contains a list of the outputs for
261      * each individual and their weight.
262      *)
263     val OutputsForInstances =
264         List.tabulate(NumIndis, fn I => (Lower + I,
265             removeNones(map (fn OutputForIndi => Vector.sub(OutputForIndi, I) )
                                OutputsForIndis) ))
266 in
267
268     (*
269      * returns a vector where each item contains tupple with the instance number
270      * and the voted output. Instances that cannot be classified are removed
271      *)
272      removeNones (map (fn (I, weightedOutputs) =>
273         (I, majorityVote(eq, weightedOutputs))) OutputsForInstances)
274 end;
275
276
277
278 (*
279  * Selects a set of individuals from the specfied kingdoms using a form of
280  * boosting. These individuals are combined through voting and the evaluation
281  * value of voting is returned.
282  * @param NumIterations the number of iterations to perform boosting
283  * @param (Fat, Slims) the kingdoms
284  *
285  * @return the eval value of the boosted committee
286  *)
287 fun boostingTestEvalValue(NumIterations, (Fat, Slims): kingdom) = let
288     (*
289      * Selects the indis using boosting
290      * The kingdoms are combined simarly to sharkpool, think it works
291      *)
292     val Indis = boosting(NumIterations, isif_remove_duplicates(
293         Population.standardIndis Fat @
294         flat_map(Population.standardIndis, Slims)))
295
296     (*
297      * Combines the elements of the two vectors into one
298      *)
299     fun makeWeightedOuput(Weight, Outputs) =
300         Vector.map (fn Output => (Weight, Output)) Outputs
301
302     (*
```

```
303         * returns a list of vectors for each individual. Each vector contains the
304         * weighted testoutputs
305         *)
306        val OutputsForIndis = map ( fn (Weight, Indi) =>
307            makeWeightedOuput(
308                Weight,
309                makeTestOutputs(#program Indi)) )
310            Indis;
311
312        (*
313         * list with all outputs for the instances that received any votes
314         *)
315        val finalOutputs = weightedTestOutputs(Spec.main_range_eq, OutputsForIndis);
316    in
317        votedTestEvalValue finalOutputs
318    end
319    (* end boosting *)
```

Appendix C

Thyroid Disease Specification

```
1  datatype age = age_NONE | age_SOME of real
2  datatype sex = sexM | sexF | sex_NONE
3  datatype on_thyroxine = on_thyroxinef | on_thyroxinet
4  datatype query_on_thyroxine = query_on_thyroxinef | query_on_thyroxinet
5  datatype on_antithyroid_medication = on_antithyroid_medicationf |
       on_antithyroid_medicationt
6  datatype sick = sickf | sickt
7  datatype pregnant = pregnantf | pregnantt
8  datatype thyroid_surgery = thyroid_surgeryf | thyroid_surgeryt
9  datatype I131_treatment = I131_treatmentf | I131_treatmentt
10 datatype query_hypothyroid = query_hypothyroidf | query_hypothyroidt
11 datatype query_hyperthyroid = query_hyperthyroidf | query_hyperthyroidt
12 datatype lithium = lithiumf | lithiumt
13 datatype goitre = goitref | goitret
14 datatype tumor = tumorf | tumort
15 datatype hypopituitary = hypopituitaryf | hypopituitaryt
16 datatype psych = psychf | psycht
17 datatype TSH_measured = TSH_measuredf | TSH_measuredt
18 datatype TSH = TSH_NONE | TSH_SOME of real
19 datatype T3_measured = T3_measuredf | T3_measuredt
20 datatype T3 = T3_NONE | T3_SOME of real
21 datatype TT4_measured = TT4_measuredf | TT4_measuredt
22 datatype TT4 = TT4_NONE | TT4_SOME of real
23 datatype T4U_measured = T4U_measuredf | T4U_measuredt
24 datatype T4U = T4U_NONE | T4U_SOME of real
25 datatype FTI_measured = FTI_measuredf | FTI_measuredt
26 datatype FTI = FTI_NONE | FTI_SOME of real
27 datatype TBG_measured = TBG_measuredf | TBG_measuredt
28 datatype referral_source = referral_sourceWEST | referral_sourceSTMW |
       referral_sourceSVHC | referral_sourceSVI | referral_sourceSVHD |
       referral_sourceother
29 datatype therapy = therapyreplacement_therapy | therapyunderreplacement |
       therapyoverreplacement | therapynegative
30
31
32 fun rconstLess( ( X, C ) : real * rconst ) : bool =
33   case C of rconst( Compl, StepSize, Current ) => realLess( X, Current )
34
35 fun tor( C : rconst ) : real =
36   case C of rconst( Compl, StepSize, Current ) => Current
37
38
39 fun f( (X0age, X1sex, X2on_thyroxine, X3query_on_thyroxine, X4on_antithyroid_medication,
        X5sick, X6pregnant, X7thyroid_surgery, X8I131_treatment, X9query_hypothyroid,
       X10query_hyperthyroid, X11lithium, X12goitre, X13tumor, X14hypopituitary, X15psych,
       X16TSH_measured, X17TSH, X18T3_measured, X19T3, X20TT4_measured, X21TT4,
       X22T4U_measured, X23T4U, X24FTI_measured, X25FTI, X26TBG_measured,
       X27referral_source ) :
40       age * sex * on_thyroxine * query_on_thyroxine * on_antithyroid_medication * sick *
             pregnant * thyroid_surgery * I131_treatment * query_hypothyroid *
             query_hyperthyroid * lithium * goitre * tumor * hypopituitary * psych *
             TSH_measured * TSH * T3_measured * T3 * TT4_measured * TT4 * T4U_measured * T4U
              * FTI_measured * FTI * TBG_measured * referral_source
41       ) : therapy =
42   raise D1
43
44
```

```
45  fun main( (X0age, X1sex, X2on_thyroxine, X3query_on_thyroxine,
        X4on_antithyroid_medication, X5sick, X6pregnant, X7thyroid_surgery, X8I131_treatment
        , X9query_hypothyroid, X10query_hyperthyroid, X11lithium, X12goitre, X13tumor,
        X14hypopituitary, X15psych, X16TSH_measured, X17TSH, X18T3_measured, X19T3,
        X20TT4_measured, X21TT4, X22T4U_measured, X23T4U, X24FTI_measured, X25FTI,
        X26TBG_measured, X27referral_source ) :
46        age * sex * on_thyroxine * query_on_thyroxine * on_antithyroid_medication * sick *
              pregnant * thyroid_surgery * I131_treatment * query_hypothyroid *
              query_hyperthyroid * lithium * goitre * tumor * hypopituitary * psych *
              TSH_measured * TSH * T3_measured * T3 * TT4_measured * TT4 * T4U_measured * T4U
               * FTI_measured * FTI * TBG_measured * referral_source
47        ) : therapy =
48     f( X0age, X1sex, X2on_thyroxine, X3query_on_thyroxine, X4on_antithyroid_medication,
          X5sick, X6pregnant, X7thyroid_surgery, X8I131_treatment, X9query_hypothyroid,
          X10query_hyperthyroid, X11lithium, X12goitre, X13tumor, X14hypopituitary, X15psych
          , X16TSH_measured, X17TSH, X18T3_measured, X19T3, X20TT4_measured, X21TT4,
          X22T4U_measured, X23T4U, X24FTI_measured, X25FTI, X26TBG_measured,
          X27referral_source )
49
50
51  %%
52
53
54
55  val Inputs = [
56  ( age_SOME ~0.3436123348017621, sexF, on_thyroxinet, query_on_thyroxinef,
        on_antithyroid_medicationf, sickt, pregnantf, thyroid_surgeryf, I131_treatmentf,
        query_hypothyroidf, query_hyperthyroidf, lithiumf, goitref, tumorf, hypopituitaryf,
        psychf, TSH_measuredt, TSH_SOME ~0.4968018566212889, T3_measuredf, T3_NONE,
        TT4_measuredt, TT4_SOME ~0.24065420560747663, T4U_measuredt, T4U_SOME
        ~0.21014492753623187, FTI_measuredt, FTI_SOME ~0.16921119592875317, TBG_measuredf,
        referral_sourceother ),
57
58  [snip...snip]
59
60  ( age_SOME ~0.32158590308370044, sexF, on_thyroxinef, query_on_thyroxinef,
        on_antithyroid_medicationf, sickf, pregnantf, thyroid_surgeryf, I131_treatmentf,
        query_hypothyroidf, query_hyperthyroidf, lithiumf, goitref, tumorf, hypopituitaryf,
        psychf, TSH_measuredt, TSH_SOME ~0.4996886763082671, T3_measuredt, T3_SOME
        ~0.3341232227488152, TT4_measuredt, TT4_SOME ~0.2266355140186916, T4U_measuredt,
        T4U_SOME ~0.18115942028985504, FTI_measuredt, FTI_SOME ~0.17430025445292618,
        TBG_measuredf, referral_sourceSVI )
61  ]
62
63
64  val Outputs = [
65  therapynegative,
66
67  [snip...snip]
68
69  therapynegative
70  ]
71
72  val All_outputs =  Vector.fromList( Outputs @ Test_outputs )
73
74  val Funs_to_use = [
75    "false", "true",
76    "realLess", "realAdd", "realSubtract", "realMultiply",
77    "realDivide", "sigmoid",
78    "tor", "rconstLess",
79    "therapyreplacement_therapy", "therapyunderreplacement", "therapyoverreplacement", "
          therapynegative"
80    ]
81
82  val Reject_funs = []
83  fun restore_transform D = D
84
85  structure Grade : GRADE =
86  struct
87
88  type grade = unit
89  val zero = ()
90  val op+ = fn(_,_) => ()
91  val comparisons = [ fn _ => EQUAL ]
92  val toString = fn _ => ""
93  val fromString = fn _ => SOME()
94
95  val pack = fn _ => ""
96  val unpack = fn _ =>()
97
98  val post_process = fn _ => ()
99
100 val toRealOpt = NONE
```

```
101 |
102 | end
103 |
104 | val Abstract_types = []
105 |
106 | fun output_eval_fun( I : int, _ , Y  ) =
107 |   if Vector.sub( All_outputs, I ) <> Y then
108 |     { numCorrect = 0, numWrong = 1, grade = () }
109 |   else
110 |     { numCorrect = 1, numWrong = 0, grade = () }
111 |
112 |
113 | val Max_output_genus_card = 4
114 | val Max_output_genus_complexity = 1.2
115 |
116 | val Max_time_limit = 1024
117 | val Time_limit_base = 1024.0
118 |
119 |
120 | val Number_of_output_attributes = 4
```

Appendix D

Pruning Specification 1

```
1  fun rconstLess( ( X, C ) : real * rconst ) : bool =
2    realLess( X, tor C )
3
4  fun log2(value: real): real =
5     realDivide(log10(value), log10(2.0))
6
7  datatype class_value = Class of int
8  datatype attribute_value = Nominal of int | Continuous of real
9  datatype split_point = NominalSplit of int * int | ContinuousSplit of int * real
10 datatype calculated_distribution = CalculatedDist of class_value * real * real
11 datatype c_tree = CLeaf of calculated_distribution * class_value
12                 | CDN of split_point * calculated_distribution * c_tree_list
13 and c_tree_list = CTreeListNil | CTreeListCons of c_tree * c_tree_list
14
15
16 fun f(curTree: c_tree): (real * c_tree) =
17 let
18   fun errorEstimate((c, n): real * real): real =
19   case realDivide(realSubtract(n, c), n) of
20     e =>
21       case tor(rconst(1, 0.1, 0.69)) of
22         z =>
23           case z * z of
24             z2 =>
25               realDivide(
26                 realAdd(
27                   realAdd(
28                     e,
29                     realDivide(z2,
30                       realMultiply(
31                         tor(rconst(1, 0.5, 2.0)),
32                         n
33                       )
34                     )
35                   ),
36                   realMultiply(z,
37                     sqrt(
38                       realAdd(
39                         realSubtract(
40                           realDivide(e, n),
41                           realDivide(
42                             e * e,
43                             n
44                           )
45                         ),
46                         realDivide(z2,
47                           realMultiply(
48                             tor(rconst(1, 0.8, 4.0)),
49                             n * n
50                           )
51                         )
52                       )
53                     )
54                   )
55                 ),
56                 realAdd(tor(rconst(1, 0.2, 1.0)), realDivide(z2, n))
57               )
```

```
58  in
59  let
60    fun pruneCTreeList treeList =
61      case treeList of
62        CTreeListNil => (tor(rconst( 0, 0.2, 0.0 )), CTreeListNil)
63        | CTreeListCons(x, xs) =>
64          case f(x) of
65            Pair2 as (errorX, prunedX) =>
66              case pruneCTreeList(xs) of
67                Pair3 as (errorXs, prunedXs) => (realAdd(errorX, errorXs),
68                CTreeListCons(prunedX, prunedXs))
69  in
70    case curTree of
71      CLeaf(dist as CalculatedDist( ClassVal1 as Class Val1, Val2, Val3 ),
72            c' as Class Val5 ) => (
73        case dist of
74          CalculatedDist(mClass as Class Val4, numInstMajorityClass, num) =>
75          (
76            realMultiply( num, errorEstimate(numInstMajorityClass ,num)
77            ),
78            curTree
79          )
80      )
81      | CDN(splitPoint,
82            dist' as CalculatedDist( ClassVal1' as Class Val1', Val2', Val3' ),
83            children) => (
84          case dist' of
85            CalculatedDist(mClass' as Class Val4', numInstMajorityClass', num') =>
86              case pruneCTreeList(children) of
87                Pair1 as (childErrorOnlyMultiplied, prunedChildren) =>
88
89                  case realDivide(childErrorOnlyMultiplied, num') of
90                    childError =>
91                      case errorEstimate(numInstMajorityClass', num') of
92                        error =>
93                          case childError < error of
94                            true =>
95                              (realMultiply(num', childError), CDN(splitPoint, dist',
                                    prunedChildren))
96                            | false =>
97                              (realMultiply(num', error), CLeaf(dist', mClass'))
98      )
99  end
100 end
101
102 fun main(curTree: c_tree): c_tree =
103   case f(curTree) of
104     (error, prunedTree) => prunedTree
105
106 %%
107
108 datatype instance = InstanceNil | InstanceCons of attribute_value * instance
109 datatype training_instance = TrainingInstance of (instance * class_value)
110 datatype data_set_info = DataSetInfo of int uncheckedArray * int
111 datatype data = DataNil | DataCons of training_instance * data
112 datatype distribution = DistributionNil | DistributionCons of real * distribution
113 datatype data_distribution = DataDistribution of data * distribution
114 datatype data_split = DataSplitNil | DataSplitCons of data_distribution * data_split
115 datatype split_point_list = SplitPointListNil | SplitPointListCons of split_point *
      split_point_list
116 datatype tree =
117   Leaf of distribution * class_value |
118   DN of split_point * distribution * tree_list
119 and tree_list = TreeListNil | TreeListCons of tree * tree_list
120
121 (* %{{{ Begin id3 *)
122 fun getClassLength(dataSetInfo: data_set_info): int =
123     case dataSetInfo of
124         DataSetInfo(attributeInfo, classLength) => classLength
125
126 fun getClassValue(trainingInstance: training_instance): class_value =
127     case trainingInstance of
128         TrainingInstance(a, classValue) => classValue
129
130 fun dataHasOnlyClass((d, c1 as Class(v1)): data * class_value): bool =
131     case d of
132         DataNil => true
133         | DataCons(x, xs) =>
134       case x of
135         TrainingInstance(i, c2 as Class(v2)) =>
136           case v1 = v2 of
137                     true => dataHasOnlyClass(xs, c1)
138                   | false => false
```

```
139
140 fun isSmall((d, number): data * int): bool =
141     case number = 0 of
142         true => (
143             case d of
144                 DataNil => true
145                 | DataCons(x, xs) => false
146         )
147         | false => (
148             case d of
149                 DataNil => true
150                 | DataCons(x, xs) => isSmall(xs, number - 1)
151         )
152
153 fun distributionSum(d: distribution): real =
154     case d of
155         DistributionNil => 0.0
156         | DistributionCons(x, xs) => realAdd(x, distributionSum(xs))
157
158 fun numInstancesMajorityClass(d: distribution): real =
159 let
160   fun helper((d, numMax): distribution * real): real =
161     case d of
162       DistributionNil => numMax
163       | DistributionCons(x, xs) =>
164         helper(
165           xs,
166           case numMax < x of
167             true => x
168             | false => numMax
169         )
170 in
171   helper(d, 0.0)
172 end
173
174 fun numInstances((d, c): distribution * class_value): real =
175 let
176   fun helper((d, index): distribution * int): real =
177     case index < 1 of
178       true =>
179         (case d of
180           DistributionNil => raise D1
181           | DistributionCons(x, xs) => x)
182       | false => (
183         case d of
184           DistributionNil => raise D1
185           | DistributionCons(x, xs) => helper(xs, index - 1)
186       )
187 in
188   case c of
189     Class(index) => helper(d, index)
190 end
191
192 fun getDistribution((dInfo, d) : data_set_info * data): distribution =
193 let
194     fun helper((d, index, classLength, count, dataLeft): data * int * int * real * data)
          : distribution =
195       case index < classLength of
196         false => DistributionNil
197         | true => (
198           case d of
199             DataNil => DistributionCons(
200                         count,
201                         helper(dataLeft, index + 1, classLength, 0.0, DataNil))
202             | DataCons(x, xs) =>
203               case getClassValue(x) of
204                 Class(v) =>
205                   case v = index of
206                     true => helper(xs, index, classLength, realAdd(count, 1.0), dataLeft
                            )
207                     | false => helper(xs, index, classLength, count, DataCons(x,
                          dataLeft))
208       )
209 in
210   case getClassLength(dInfo) of
211     cLength => helper(d, 0, cLength, 0.0, DataNil)
212 end
213
214 fun majorityClass(d: distribution): class_value =
215 let
216     fun helper((dist, bestIndex, bestValue, count): distribution * int * real * int):
          class_value =
217         case dist of
```

```
218 |             DistributionNil => Class(bestIndex)
219 |             | DistributionCons(x, xs) =>
220 |                 case realLess(x, bestValue) of
221 |                     true => helper(xs, bestIndex, bestValue, count + 1)
222 |                     | false => helper(xs, count, x, count + 1)
223 | in
224 |     helper(d, 0, ~1000.0, 0)
225 | end
226 |
227 | fun partEntropy((ratio): real): real =
228 |   realSubtract(0.0,
229 |       realMultiply(
230 |           ratio,
231 |           log2(ratio)
232 |       )
233 | )
234 |
235 | fun entropy((dist, total): distribution * real): real =
236 |   case dist of
237 |       DistributionNil => 0.0
238 |       | DistributionCons(positive, xs) =>
239 |         realAdd(
240 |           case realLess(positive, 0.0) of
241 |           true => 0.0
242 |           | false => (
243 |             case realEqual(positive, 0.0) of
244 |               true => 0.0
245 |               | false =>  partEntropy(realDivide(positive, total))
246 |             )
247 |         ,
248 |             entropy(xs, total)
249 |         )
250 |
251 | fun almostInfoGain(dataSplits: data_split): real =
252 |     case dataSplits of
253 |         DataSplitNil => 0.0
254 |         | DataSplitCons(x, xs) =>
255 |             case x of
256 |               DataDistribution(d, dist) =>
257 |                 case distributionSum(dist) of
258 |                   len => realAdd(realMultiply(len, entropy(dist, len)), almostInfoGain(
                              xs))
259 |
260 | fun getAttributeIndex((splitPoint): split_point): int =
261 |   case splitPoint of
262 |       NominalSplit(index, len) => index
263 |         | ContinuousSplit(index, realValue) => index
264 |
265 | fun getAttributeLength((splitPoint): split_point): int =
266 |   case splitPoint of
267 |       NominalSplit(index, len) => len
268 |         | ContinuousSplit(index, realValue) => 2
269 |
270 | fun splitPointEq((sp1, sp2):split_point * split_point): bool =
271 |   case sp1 of
272 |     ContinuousSplit(i1, r1) => (
273 |       case sp2 of
274 |         NominalSplit(i2, nl2) => false
275 |         | ContinuousSplit(i2, r2) => (
276 |           case i1 = i2  of
277 |             true => realEqual(r1, r2)
278 |             | false => false)
279 |       )
280 |     | NominalSplit(i1, nl1) => (
281 |       case sp2 of
282 |         NominalSplit(i2, nl2) =>
283 |           (case i1 = i2 of
284 |             true => nl1 = nl2
285 |             | false => false
286 |           )
287 |         | ContinuousSplit(i2, r2) => false
288 |                 )
289 |
290 | fun getSplitPoints((splitPoint, splitPoints, d): split_point * split_point_list * data):
        split_point_list =
291 |   case splitPoints of
292 |       SplitPointListNil => SplitPointListNil
293 |       | SplitPointListCons(x, xs) =>
294 |           case splitPointEq(x, splitPoint)  of
295 |               true => getSplitPoints(splitPoint, xs, d)
296 |               | false => SplitPointListCons(x, getSplitPoints(splitPoint, xs, d))
297 |
298 | fun splitPointListToDataSetInfo((splitPointList, classLength): split_point_list * int):
```

```
    data_set_info =
let
  fun helper((sps, i): split_point_list * int): int uncheckedArray =
    case sps of
      SplitPointListNil => uncheckedArray(i, ~1)
      | SplitPointListCons(x, xs) => uncheckedArrayUpdate(helper(xs, i + 1), i,
          getAttributeLength x)
in
  DataSetInfo(helper(splitPointList, 0), classLength)
end

fun getSplitPointIndexWithAssumptions((splitPoint, inst): split_point * instance):int =
    case inst of
        InstanceNil => raise D1
        | InstanceCons(x, xs) =>
            case splitPoint of
                NominalSplit(attributeIndex, nominalList) => (
                    case x of
                        Nominal(value) => value
                        | Continuous(value) => raise D1)
            | ContinuousSplit(attributeIndex, splitValue) => (
                case x of
                    Nominal(value) => raise D1
                    | Continuous(value) =>

                        case value < splitValue of
                            true => 0
                            | false => 1)

fun split((dInfo, fullData, d, splitPoint): data_set_info * data * data * split_point):
    data_split =
let
    fun helper((splitPoint, dInfo, fullData, d, attv, fullDataLeft, fullDataRight,
        dataRight):
      split_point * data_set_info * data * data * int * data * data * data): data_split
          =

        case d of
            DataNil => (
              case attv < (case splitPoint of ContinuousSplit(x,v) => 2 | NominalSplit(x
                  , len) => len) of
                true =>
                    DataSplitCons(
                      DataDistribution(
                        fullDataLeft,
                        getDistribution(dInfo, fullDataLeft)),
                      helper(splitPoint, dInfo, fullDataRight, dataRight, attv + 1,
                          DataNil, DataNil, DataNil))
              | false => DataSplitNil
            )
            | DataCons(x, xs) =>
              case fullData of
                  DataNil => raise D2
                    | DataCons(fx, fxs) =>
                  (case x of
                        TrainingInstance(inst, class) =>
                            case attv = getSplitPointIndexWithAssumptions(splitPoint,
                                inst) of
                            true => helper(splitPoint, dInfo, fxs, xs, attv,
                              DataCons(fx, fullDataLeft), fullDataRight, dataRight)
                            | false => helper(splitPoint, dInfo, fxs, xs, attv,
                              fullDataLeft, DataCons(fx, fullDataRight), DataCons(x,
                                  dataRight))
                  )
in
    helper(splitPoint, dInfo, fullData, d, 0, DataNil, DataNil, DataNil)
end

fun removeFirstAttribute(d: data): data =
    case d of
        DataNil => DataNil
        | DataCons(x, xs) =>
            case x of
                TrainingInstance(i, c) =>
                    case i of
                        InstanceNil => raise D1
                        | InstanceCons(at, ats) =>
                            DataCons(
                                TrainingInstance(ats, c),
                                removeFirstAttribute(xs))

fun findMax((splitPoints, dInfo, fullData): split_point_list * data_set_info * data):
    split_point * real * data_split =
```

```
372 let
373   fun helper((splitPoints, splitPointIndex, partialData):
374     split_point_list * int * data): split_point * real * data_split =
375     case splitPoints of
376       SplitPointListNil => raise D1
377       | SplitPointListCons(x, xs) =>
378         case (case x of
379           NominalSplit(aIndex, l) => aIndex
380           | ContinuousSplit(aIndex, splitValue) => aIndex) = splitPointIndex of
381           true => (
382             case split(dInfo, fullData, partialData, x) of
383               dataSplits =>
384                 case almostInfoGain(dataSplits) of
385                   fitness =>
386                     case xs of
387                       SplitPointListNil => (x, fitness, dataSplits)
388                       | SplitPointListCons(x',xs') => (
389
390                         case helper(xs, splitPointIndex, partialData) of
391                           (bestSplitPoint, bestFitness, bestDataSplits) =>
392                             case fitness < bestFitness of
393                               true => (x, fitness, dataSplits)
394                               | false => (bestSplitPoint, bestFitness,
395                             bestDataSplits)
396                       )
397           )
398
399           | false => helper(splitPoints, splitPointIndex + 1, removeFirstAttribute(
                partialData))
400 in
401   case splitPoints of
402     SplitPointListNil => raise D1
403     | SplitPointListCons(x,xs) => helper(splitPoints, 0, fullData)
404 end
405
406 fun makeTree((splitPoints, minimumExamples, dInfo, d, dist, parentDist):
407       split_point_list * int * data_set_info * data * distribution * distribution): tree
            =
408 let
409     fun makeTreeList((splitPoints, splitPoint, dataSplits, parentDist):
410           split_point_list * split_point * data_split * distribution): tree_list =
411         case dataSplits of
412             DataSplitNil => TreeListNil
413             | DataSplitCons(x, xs) =>
414               case x of
415                 DataDistribution(d, dist) =>
416                 TreeListCons(
417                     makeTree(
418                         getSplitPoints(splitPoint, splitPoints, d),
419                         minimumExamples,
420                         dInfo,
421                         d,
422                         dist,
423                         parentDist),
424                     makeTreeList(
425                         splitPoints,
426                         splitPoint,
427                         xs,
428                         parentDist))
429 in
430
431     case isSmall(d, minimumExamples) of
432         true => Leaf(dist, majorityClass parentDist)
433         | false =>
434             case d of
435                 DataNil => Leaf(dist, majorityClass parentDist)
436                 | DataCons(x, xs) =>
437                     case dataHasOnlyClass(d, getClassValue(x)) of
438                         true => Leaf(dist, getClassValue(x))
439                         | false =>
440                             case splitPoints of
441
442                                 SplitPointListNil => Leaf(dist, majorityClass dist)
443                                 | SplitPointListCons(ignoreX, ignoreXs) =>
444                                     case findMax(splitPoints, dInfo, d) of
445                                         (splitPoint, fitness, dataSplits) =>
446                                             DN(
447                                                 splitPoint,
448                                                 dist,
449                                                 makeTreeList(
450                                                     splitPoints,
451                                                     splitPoint,
452                                                     dataSplits,
```

```
453                                                    dist
454                                                ))
455 end
456
457 fun createTree((splitPoints, dInfo, d): split_point_list * data_set_info * data): tree =
458   case getDistribution(dInfo, d) of
459     dist =>  makeTree(splitPoints, 0, dInfo, d, dist, dist)
460 (* %}}} End id3 functions *)
461
462 (* %{{{ Begin util functions *)
463 fun println(string) = print(string ^ "\n")
464
465 fun partitioni f l =
466 let
467   fun helper(i, nil) = (nil, nil)
468   | helper(i, x :: xs) = let
469     val (ll, rl) = helper(i+1, xs)
470   in
471     if f(i, x) then (x :: ll, rl) else (ll, x :: rl)
472   end
473 in
474   helper(0, l)
475 end
476
477 fun vectorToList v = Vector.foldr (fn (x, l) => x :: l) [] v
478
479 fun stringNTimes(s, n) = String.concat(List.tabulate(n, fn x => s))
480
481 fun rangeInt(from, to, step): int list =
482   if from < to then
483     from :: rangeInt(from + step, to, step)
484   else
485     nil
486
487 fun arrayToList(a) = Array.foldr (fn (x,l) => x :: l) nil a
488
489 fun listFoldri f init l =
490 let
491   fun helper(i, nil) = init
492   | helper(i, x :: xs) =
493     f(i,x, helper(i+1, xs))
494 in
495   helper(0, l)
496 end
497
498 fun vectorRemoveIndex(v, i) =
499   VectorSlice.concat([
500     VectorSlice.slice(v, 0, SOME i),
501     VectorSlice.slice(v, i + 1, NONE)
502   ])
503 (* %}}} end util functions *)
504
505 (* %{{{ Begin conversion functions *)
506 fun listToInstance nil = InstanceNil
507 | listToInstance(x :: xs) = InstanceCons(x, listToInstance xs)
508
509 local
510     fun toData convertToInstance data =
511     let
512
513         fun helper nil = DataNil
514         | helper((instance, classValue) :: xs) = DataCons(
515                 TrainingInstance(
516                    convertToInstance instance,
517                     classValue),
518                 helper(xs))
519     in
520         helper(data)
521     end
522
523     fun vectorToInstance( x: attribute_value vector) =
524       listToInstance (vectorToList x)
525 in
526     val listAttributeVectorClassValueToData  = (toData vectorToInstance)
527 end
528
529 fun listToSplitPointList nil = SplitPointListNil
530     | listToSplitPointList(x :: xs) = SplitPointListCons(x, listToSplitPointList(xs))
531
532 fun splitPointDecreaseIndex(sp) =
533     case sp of
534       NominalSplit(i, numAttributeValues) => NominalSplit(i -1, numAttributeValues)
535       | ContinuousSplit(i, v) => ContinuousSplit(i -1, v)
```

```
536
537 fun splitPointsToDataSetInfo(splitPoints, numClasses) =
538 let
539   val len = List.length(splitPoints)
540   fun helper(a, nil) = a
541   | helper(a, sp :: sps) =
542     case getAttributeIndex(sp) of
543       index =>
544         case index >= 0 andalso index < len of
545           true => (uncheckedArrayUpdate(a, index, getAttributeLength(sp));helper(a,sps))
546           | false => raise D1
547
548 in
549   DataSetInfo(helper(uncheckedArray(len, ~1), splitPoints), numClasses)
550 end
551
552 fun distToCalculatedDist(dist) =
553   CalculatedDist(
554     majorityClass dist,
555     numInstancesMajorityClass dist,
556     distributionSum dist
557   )
558
559 fun treeToCTree(
560       Leaf(dist, classValue)) = CLeaf(distToCalculatedDist(dist), classValue)
561 | treeToCTree (
562     DN(
563       splitPoint,
564       dist,
565       children)) =
566         CDN(
567           splitPoint,
568           distToCalculatedDist dist,
569           treeListToCTreeList children
570         )
571 and treeListToCTreeList(TreeListNil) = CTreeListNil
572 | treeListToCTreeList(TreeListCons(x, xs)) =
573     CTreeListCons(
574       treeToCTree x,
575       treeListToCTreeList(xs)
576     )
577 (* %}}} End conversion functions *)
578
579 (* %{{{ Begin classification *)
580 fun splitPointIndexInstanceVector((splitPoint, inst): split_point * attribute_value
      vector): int =
581   case splitPoint of
582     NominalSplit(index, len) => (
583       case Vector.sub(inst, index) of
584         Nominal(value) => value
585         | _ => raise D1
586     )
587     | ContinuousSplit(index, splitValue) => (
588       case Vector.sub(inst, index) of
589         Continuous(value) => if value < splitValue then 0 else 1
590         | _ => raise D1
591     )
592
593 fun subTreeList((index, treeList):  int * tree_list): tree =
594     case treeList of
595         TreeListNil => raise D2
596         | TreeListCons(x, xs) =>
597             case 0 = index of
598                 true => x
599                 | false => subTreeList(index - 1, xs)
600
601 fun classify((tr, inst): tree * attribute_value vector):class_value =
602   case tr of
603         Leaf(dist, classValue) => classValue
604         | DN(splitPoint, dist, treeList) =>
605           classify(
606             subTreeList(
607               splitPointIndexInstanceVector(splitPoint,  inst),
608               treeList),
609             inst
610           )
611
612 fun subCTreeList((index, treeList):  int * c_tree_list): c_tree =
613     case treeList of
614         CTreeListNil => raise D2
615         | CTreeListCons(x, xs) =>
616             case 0 = index of
617                 true => x
```

```
618 |                 | false => subCTreeList(index - 1, xs)
619 |
620 | fun classifyCTree((tr, inst): c_tree * attribute_value vector):class_value =
621 |   case tr of
622 |         CLeaf(dist, classValue) => classValue
623 |         | CDN(splitPoint, dist, treeList) =>
624 |             classifyCTree(
625 |                 subCTreeList(
626 |                   splitPointIndexInstanceVector(splitPoint,  inst),treeList
627 |                 ),
628 |                 inst
629 |               )
630 | (* %}}} End classification *)
631 |
632 | (* %{{{ Begin random *)
633 | fun getRandomReal() =
634 |     Real.fromLargeInt(Word.toLargeInt(MLton.Random.rand())) / Math.pow(2.0, Real.fromInt
          (Word.wordSize))
635 |
636 | fun getRandomRealX() =
637 | let
638 |     val r = (Real.fromLargeInt(Word.toLargeIntX(MLton.Random.rand()))) / Math.pow(2.0,
          Real.fromInt(Word.wordSize))
639 | in
640 |  (if r < 0.0 then 1.0 + r else r)
641 | end
642 |
643 | fun getRandomGaussian() =
644 | let
645 |     val x1 = 2.0 * getRandomReal() - 1.0
646 |     val x2 = 2.0 * getRandomReal() - 1.0
647 |     val w = x1 * x1 + x2 * x2
648 | in
649 |     if w >= 1.0 orelse Real.==(w, 0.0) then
650 |         getRandomGaussian()
651 |     else
652 |         x1 * Math.sqrt( (~2.0 * Math.ln( w ) ) / w )
653 | end
654 |
655 | fun getRandomInt(to) =
656 |   Real.floor(Real.fromInt(to) * getRandomReal())
657 | (* %}}} End random *)
658 |
659 | fun sumCorrectAndWrong classify (tree, testInstances) =
660 | let
661 |   fun addCorrectOrWrong((instance, classValue), (correct, wrong))=
662 |     if classify(tree, instance) = classValue then (correct+1, wrong) else (correct,
          wrong+1)
663 | in
664 |   List.foldr addCorrectOrWrong (0,0) testInstances
665 | end
666 |
667 | (* %{{{ Begin synthetic data *)
668 | exception CustomE of string
669 |
670 | fun getTripplePermutations(oneList, twoList, threeList) =
671 |   List.concat(List.concat(
672 |     List.map
673 |       (fn one =>
674 |           List.map
675 |             (fn two =>
676 |               List.map
677 |                 (fn three =>
678 |                   (one, two, three)
679 |                 )
680 |               threeList
681 |             )
682 |           twoList
683 |         )
684 |         oneList
685 |   ))
686 |
687 | fun shuffleArray(a) =
688 | let
689 |   val len = Array.length a
690 | in
691 |     Array.appi
692 |       (fn (index, _) =>
693 |       let
694 |
695 |         val rIndex = index + getRandomInt(len - index)
696 |         val v = Array.sub(a, index)
697 |         val rv = Array.sub(a, rIndex)
```

```
698         val _ = Array.update(a, index, rv)
699         val _ = Array.update(a, rIndex, v)
700       in
701         ()
702       end)
703     a
704 end
705
706 fun attributeIndexValuePermutations(postfix, splitPoints): (int * attribute_value) list
      list =
707 let
708   fun addPermutationsForSplitPoint(sp, l) =
709   let
710     val (attribIndex, attribLength) = (case sp of
711                                          NominalSplit(aIndex, aLength) => (aIndex,
                                               aLength)
712                                        | ContinuousSplit(aIndex, _) => (aIndex, 2)
713                                       )
714   in
715     List.concat(List.tabulate(attribLength,
716         fn attribValue => List.map (fn x => (attribIndex, Nominal(attribValue)) :: x) l)
              )
717   end
718 in
719   (List.foldr addPermutationsForSplitPoint [postfix] splitPoints)
720 end
721
722 fun permutations(splitPoints, handledSplitPoints, classValue, numAttributes): (
      attribute_value vector * class_value) list =
723 List.map
724   (fn permutation =>
725     let
726       val a = Array.array(numAttributes, Nominal(~1))
727       val _ = List.app (fn (ai, av) => Array.update(a, ai, av)) permutation
728     in
729       (Array.vector a, Class(classValue))
730     end
731   )
732   (attributeIndexValuePermutations(handledSplitPoints, splitPoints))
733
734 fun removeSplitPoint(splitPoints, splitPoint) =
735   List.filter (fn x => not( splitPointEq(splitPoint,x))) splitPoints
736
737 fun listToTrainTest (data, fraction, stratified) =
738 let
739   val _ = (fraction >= 0.0 andalso fraction <= 1.0)
740             orelse raise CustomE("fraction must be between 0 and 1.0")
741   val breakPoint = Real.floor(fraction * Real.fromInt(List.length data))
742 in
743   listFoldri
744     (fn (i, x, (ll, lr) ) => if i < breakPoint then (x :: ll, lr) else (ll, x:: lr))
745     ([],[])
746      data
747 end
748
749 fun createBinarySplitPoints(numAttributes) =
750     List.tabulate(numAttributes,
751       fn i => NominalSplit(i, 2)
752 )
753
754 fun intPermutations(lengths) =
755 let
756   fun addToPermutations(x, ll) =
757     List.concat(List.tabulate(x,
758         fn value => List.map (fn l => value :: l) ll))
759 in
760   (List.foldr addToPermutations [[]] lengths)
761 end
762
763 fun randomWeights(length) =
764   List.tabulate(length, fn x => getRandomReal() - 0.5)
765
766 fun randomFunctions(numWeights, numClasses) =
767   List.tabulate(numClasses, fn x => randomWeights(numWeights))
768
769 fun maxIndex(values: (real) list) = let
770 val (index, value) =
771   listFoldri (fn (ix, vx, (iy, vy)) => if vx >= vy then (ix, vx) else (iy, vy)) (~1, ~
        Real.maxFinite) values
772
773 val _ = (index <> ~1) orelse raise CustomE("index is ~1, and this should not happen")
774 in
775   index
```

```
776 end
777
778 fun functionValue(weights, attributes) =
779   ListPair.foldr (fn (w, a, sum) => (w * Real.fromInt(a)) + sum) 0.0 (weights,
          attributes)
780
781 fun splitPointsToLengthList(splitPoints) = List.map (fn x => getAttributeLength(x))
        splitPoints
782
783 fun addBiasToAttributes functions = List.map (fn weights => 1 :: weights) functions
784
785 fun generateDataSet(splitPoints, functions: real list list, withBias) =
786 let
787   val permutations = intPermutations( splitPointsToLengthList splitPoints )
788   val permutations = if withBias then addBiasToAttributes permutations else permutations
789
790   val attributeValuesList = List.map (fn permutation => List.map (fn x => Nominal(x))
          permutation) permutations
791   val classes = List.map
792                   (fn attributes =>
793                     maxIndex(
794                       List.map (
795                         fn weights => functionValue(weights, attributes))
796                       functions
797                     )
798                   )
799                   permutations
800 in
801   ListPair.map (fn (a, v) => (Vector.fromList a, Class(v))) (attributeValuesList,
          classes)
802 end
803
804 fun linesCreateInputOutput(splitPoints, numClasses, withBias)=
805   let
806     val numWeights = if withBias then List.length splitPoints + 1 else List.length
            splitPoints
807     val functions = randomFunctions(numWeights, numClasses)
808     val data = generateDataSet(splitPoints, functions, withBias)
809   in
810     (functions, data)
811   end
812
813 fun removeSplitPointWithArranging(splitPoints: split_point list, index) =
814   listFoldri
815     (fn (i, x, y) =>
816       if i > index then
817         splitPointDecreaseIndex(x) :: y
818       else if i = index then y
819       else x :: y
820     )
821     [] splitPoints
822
823 fun removeRandomAttribute(splitPoints, data) =
824 let
825   val numAttributes = List.length splitPoints
826   val index = getRandomInt(numAttributes)
827   val splitPoints = removeSplitPointWithArranging(splitPoints, index)
828   val _ = (numAttributes = (List.length splitPoints + 1)) orelse
829               raise CustomE("Did_not_remove_attribute")
830   val data = List.map (fn (v,c) => (vectorRemoveIndex(v, index),c)) data
831
832 in
833   (splitPoints, data)
834 end
835
836 fun removeNRandomAttributes(splitPoints, data, n) =
837   List.foldr
838     (fn (_, (splitPoints, data)) =>
839
840       removeRandomAttribute(splitPoints, data)
841     )
842     (splitPoints, data)
843     (List.tabulate(n, fn x => x))
844
845 fun manipulateDataSetWithTrain(
846   (splitPoints,
847   data, numClasses),
848   (amountTraining,
849     amountOfTrainingToTraining,
850     numAttributesToRemove)) =
851 let
852     val testLimit = 3000
853     val (splitPoints, data) = removeNRandomAttributes(splitPoints, data,
```

```
        numAttributesToRemove)
854
855     val (train, test) = listToTrainTest(data, amountTraining, false)
856
857     val train = List.take(train, Real.floor(Real.fromInt(List.length train) *
            amountOfTrainingToTraining) )
858     val test = scramble test
859     val test = if List.length test > testLimit then List.take(test, testLimit) else test
860
861     val dInfo = splitPointsToDataSetInfo(splitPoints, numClasses)
862
863     val ttree = createTree(listToSplitPointList splitPoints, dInfo, (
            listAttributeVectorClassValueToData train))
864
865     val (trainC, trainW) = sumCorrectAndWrong classify(ttree, train)
866     val trainSize = List.length train
867     val (testC, testW) = sumCorrectAndWrong classify (ttree, test)
868     val testSize = List.length test
869     val input = ttree
870     val output = ((trainC, trainW, trainSize), (testC, testW, testSize), train, test)
871 in
872   (input, output)
873 end
874
875 fun manipulateDataSet(args) =
876   case manipulateDataSetWithTrain(args) of
877     (input, ((trainC, trainW, trainSize), (testC, testW, testSize), train, test)) =>
878       (input, ((trainC, trainW, trainSize), (testC, testW, testSize), test))
879
880 fun createAndManipulateDataset
881     createDataSet
882     manipulateDataSet
883     manipulationArgsList
884     createNewModelEachTime
885     numModels =
886       List.tabulate(numModels,
887         (fn x =>
888         let
889           val dataSetResult = createDataSet()
890         in
891           List.map
892             (fn manipulationArgs =>
893             let
894               val dataSetResult = if createNewModelEachTime then createDataSet() else
                        dataSetResult
895             in
896               manipulateDataSet(dataSetResult, manipulationArgs)
897             end)
898           manipulationArgsList
899         end
900         ))
901
902 fun createSynteticInputsOutputs
903       createDataSet
904       manipulateDataSet
905       modelArgsList
906       manipulationArgsList
907       numModels =
908   ListPair.unzip(List.concat(List.concat(
909     List.map
910       (fn modelArgs =>
911           (createAndManipulateDataset
912             (createDataSet modelArgs)
913             manipulateDataSet
914             manipulationArgsList
915             false
916             numModels
917           )
918       )
919     modelArgsList )))
920
921 fun genDataFromFunctionModel(splitPoints, numClasses, _) =
922   #2(linesCreateInputOutput( splitPoints, numClasses, false))
923
924 fun genModel genData (splitPoints, numClasses, depth) () =
925 let
926     val withBias = true
927     val (data) = genData(splitPoints, numClasses, depth)
928     val a = Array.fromList data
929     val _ = shuffleArray a
930 in
931   (splitPoints, arrayToList a, numClasses)
932 end
```

```
933 |
934 | fun manipulateDataSetWithCTree(args) =
935 |   case manipulateDataSetWithTrain(args) of
936 |     (input, ((trainC, trainW, trainSize), (testC, testW, testSize), train, test)) =>
937 |       (treeToCTree input, ((trainC, trainW, trainSize), (testC, testW, testSize), test))
938 |
939 | fun getDataSetArguments() =
940 | let
941 |   val splitPointsList = List.map (fn x => createBinarySplitPoints x)
942 | [10, 11, 12]
943 |   val numClassesList = [2]
944 |   val depthList = [~1]
945 | in
946 |   getTripplePermutations(splitPointsList, numClassesList, depthList)
947 | end
948 |
949 | fun getManipulationArguments() =
950 | let
951 |
952 |   val amountTrainingList = [0.5]
953 |   val amountOfTrainingToTrainingList = [0.2, 0.25, 0.3, 0.35]
954 |
955 |   val numAttributeToRemoveList = rangeInt(0, 5, 1)
956 | in
957 |   getTripplePermutations(amountTrainingList, amountOfTrainingToTrainingList,
      |       numAttributeToRemoveList)
958 | end
959 |
960 | val createInputsOutputs
961 |    = (createSynteticInputsOutputs
962 |       (genModel genDataFromFunctionModel)
963 |       (manipulateDataSetWithCTree)
964 |       (getDataSetArguments())
965 |       (getManipulationArguments())
966 |       )
967 | (* %}}} End synthetic data *)
968 |
969 | val num = 16
970 | val (Test_inputs, Test_outputs) = (createInputsOutputs 20)
971 | val (Validation_inputs, Validation_outputs) = (Test_inputs, Test_outputs)
972 | val (Inputs, Outputs) = (createInputsOutputs num)
973 | val All_outputs =  Vector.fromList( Outputs @ Test_outputs )
974 |
975 | val Funs_to_use = [
976 |   "false", "true", "=",
977 |   "realLess", "realAdd", "realSubtract", "realMultiply",
978 |   "realDivide", "tanh","tor", "rconstLess", "sqrt", "ln",
979 |   "CTreeListNil", "CTreeListCons",
980 |   "CLeaf", "CDN",
981 |   "CalculatedDist"
982 |   ]
983 |
984 | val Reject_funs = []
985 | fun restore_transform D = D
986 |
987 | structure Grade : GRADE =
988 | struct
989 |
990 | type grade = unit
991 | val zero = ()
992 | val op+ = fn(_,_) => ()
993 | val comparisons = [ fn _ => EQUAL ]
994 | val toString = fn _ => ""
995 | val fromString = fn _ => SOME()
996 |
997 | val pack = fn _ => ""
998 | val unpack = fn _ =>()
999 |
1000 | val post_process = fn _ => ()
1001 |
1002 | val toRealOpt = NONE
1003 |
1004 | end
1005 |
1006 | val Abstract_types = [ "split_point" ]
1007 |
1008 | val sumCAndW = sumCorrectAndWrong classifyCTree
1009 |
1010 | fun output_eval_fun(I: int, _, prunedTree: c_tree) =
1011 | let
1012 |   val ((treeTrainC, treeTrainW, treeTrainSize),
1013 |       (treeTestC, treeTestW, treeTestSize),
1014 |       testData) = Vector.sub(All_outputs, I)
```

```
1015 |
1016 |   val (prunedTreeC, prunedTreeW) = sumCAndW(prunedTree, testData)
1017 |   val cImp = prunedTreeC - treeTestC
1018 |   val c = Real.floor((Real.fromInt(cImp) / Real.fromInt(treeTestC)) * 10000.0)
1019 | in
1020 |   {numCorrect = prunedTreeC, numWrong = prunedTreeW, grade = ()}
1021 | end
1022 |   handle Ex =>
1023 |     {numCorrect = 0, numWrong = 100000, grade = ()}
1024 |
1025 | fun calcDistEq( CalculatedDist( X1, Y1, Z1 ), CalculatedDist( X2, Y2, Z2 ) ) =
1026 |       X1 = X2 andalso Real.==( Y1, Y2 ) andalso Real.==( Z1, Z2 )
1027 |
1028 | fun split_point_eq( NominalSplit X, NominalSplit Y ) = X = Y
1029 | fun split_point_eq( ContinuousSplit X, ContinuousSplit Y ) =
1030 |       #1 X = #1 Y andalso Real.==( #2 X, #2 Y )
1031 |   | split_point_eq( _, _ ) = false
1032 |
1033 | fun c_tree_eq( CLeaf( CalcDist1, ClassVal1 ), CLeaf( CalcDist2, ClassVal2 ) ) =
1034 |       calcDistEq( CalcDist1, CalcDist2 ) andalso ClassVal1 = ClassVal2
1035 |   | c_tree_eq( CDN( P1, Dist1, Xs1 ), CDN( P2, Dist2, Xs2 ) ) =
1036 |       split_point_eq( P1, P2 ) andalso
1037 |       calcDistEq( Dist1, Dist2 ) andalso
1038 |       c_tree_list_eq( Xs1, Xs2 )
1039 |   | c_tree_eq( _, _ ) = false
1040 |
1041 | and c_tree_list_eq( CTreeListNil, CTreeListNil ) = true
1042 |   | c_tree_list_eq( CTreeListCons( X1, Xs1 ), CTreeListCons( Y1, Ys1 ) ) =
1043 |       c_tree_eq( X1, Y1 ) andalso c_tree_list_eq( Xs1, Ys1 )
1044 |
1045 | val Max_output_genus_card = 4
1046 | val Max_output_genus_complexity = 1.2
1047 |
1048 | val Max_time_limit = 262144
1049 | val Time_limit_base = 262144.0
1050 | val Max_syntactic_complexity = 500.0
1051 |
1052 | val main_range_eq = c_tree_eq
1053 |
1054 | val Number_of_output_attributes = 1
```

Appendix E

Synthesized Program for Pruning Specification 1

```
fun f curTree =
  let
    fun g173652( V173653, V173654 ) =
      realDivide(
        realAdd(
          tanh(
            tanh(
              tanh(
                realDivide(
                  realSubtract( V173654, V173653 ),
                  V173654
                  )
                )
              )
            ),
          sqrt( tanh( tanh( sqrt( V173654 ) ) ) )
          ),
        tanh( sqrt( sqrt( V173653 ) ) )
        )
  in
    case curTree of
      CLeaf(
        V1ED3DC as
          CalculatedDist(
            V1ED3DD as Class( V1ED3DE ),
            V1ED3DF,
            V1ED3E0
            ),
        V1ED3E1 as Class( V1ED3E2 )
        ) =>


        (realMultiply( V1ED3E0, g173652( V1ED3DF, V1ED3E0 ) ),
          curTree
          )
    | CDN(
        V1ED3E7,
        V1ED3E8 as
          CalculatedDist(
            V1ED3E9 as Class( V1ED3EA ),
            V1ED3EB,
            V1ED3EC
            ),
        V1ED3ED
        ) =>
    case
      let
        fun g15001C3 V15001C4 =
          case V15001C4 of
            CTreeListNil =>
              (
                tor( rconst( 0, 0.25, 0.40268806668952906 ) ),
                CTreeListNil
```

```
54 |               )
55 |           | CTreeListCons( V15001C5, V15001C6 ) =>
56 |           case f( V15001C5 ) of
57 |             V15001C7 as ( V15001C8, V15001C9 ) =>
58 |           case g15001C3( V15001C6 ) of
59 |             V15001CA as ( V15001CB, V15001CC ) =>
60 |               (
61 |                 realAdd( V15001C8, V15001CB ),
62 |                 CTreeListCons( V15001C9, V15001CC )
63 |                 )
64 |       in
65 |         g15001C3( V1ED3ED )
66 |       end    of
67 |       V15001CD as ( V15001CE, V15001CF ) =>
68 |     case
69 |       realLess(
70 |         realDivide( V15001CE, V1ED3EC ),
71 |         g173652( V1ED3EB, V1ED3EC )
72 |         )    of
73 |       true => ( V15001CE, CDN( V1ED3E7, V1ED3E8, V15001CF ) )
74 |     | false => f( CLeaf( V1ED3E8, V1ED3E9 ) )
75 | end
```

Appendix F

Pruning Specification 2

The second pruning specification is similar to the first pruning specification, and major parts of both specifications are identical. Thus, to save space, some of these parts are replaced with [snip...snip].

```
1  fun rconstLess( ( X, C ) : real * rconst ) : bool =
2    realLess( X, tor C )
3
4  fun log2(value: real): real =
5     realDivide(log10(value), log10(2.0))
6
7  datatype class_value = Class of int
8  datatype attribute_value = Nominal of int | Continuous of real
9  datatype split_point = NominalSplit of int * int | ContinuousSplit of int * real
10 datatype calculated_distribution = CalculatedDist of class_value * real * real * real
11 datatype c_tree = CLeaf of calculated_distribution
12                 | CDN of split_point * calculated_distribution * c_tree_list
13 and c_tree_list = CTreeListNil | CTreeListCons of c_tree * c_tree_list
14
15
16 fun f(curTree: c_tree): (real * c_tree) =
17 let
18     fun errorEstimate((sc, sn, sN): real * real * real): real =
19       case sn * tor(rconst(0, 10.0, 100.0)) of
20         n =>
21         case (n -( sc * tor(rconst(0, 0.2, 2.0)) * n)) of
22           e =>
23             case e < tor(rconst(0, 0.1, 1.0)) of
24               true =>
25                 (case n *
26                   (
27                     tor(rconst(0, 0.1, 1.0))
28                     - pow(tor(rconst(0, 0.025, 0.25)), tor(rconst(0, 0.1, 1.0)) / n)
29                   ) of
30                   base =>
31                     case tor(rconst(0, 0.01, 0.0)) < e of
32                       true => base + (e * (errorEstimate((n-tor(rconst(0, 0.1,
                                1.000000001)))/(tor(rconst(0, 0.2, 2.0)) * n), sn, sN)) - base
                                )
33                       | false => base
34                   )
35               | false =>
36                 case n < e + tor(rconst(0, 0.05, 0.5)) of
37                   true => (
38                     case tor(rconst(0, 0.01, 0.0)) < n - e of
39                       true => n-e
40                       | false => tor(rconst(0, 0.01, 0.0))
41                   )
42                   | false =>
43                   case (e  + tor(rconst(0, 0.05, 0.5))) / n of
44                     errorRate =>
45                       case tor(rconst(0, 0.111, 0.674489751129221500)) of
46                         z =>
```

```
47                          case (errorRate + (z * z) / (tor(rconst(0, 0.2, 2.0)) * n) +
48                          z * sqrt((errorRate / n) -
49                          (errorRate * errorRate / n) +
50                        (z * z / (tor(rconst(0, 0.4, 4.0)) * n * n)))) /
51                    (tor(rconst(0, 0.1, 1.0)) + (z * z) / n)
52                  of r => ( (r * n) - e)
53
54  in
55      let
56        fun pruneCTreeList(treeList:c_tree_list):(real * c_tree_list) =
57          case treeList of
58            CTreeListNil => (tor(rconst( 0, 0.01, 0.0 )), CTreeListNil)
59            | CTreeListCons(x, xs) =>
60              case f(x) of
61                Pair2 as (errorX, prunedX) =>
62                  case pruneCTreeList(xs) of
63                    Pair3 as (errorXs, prunedXs) => (realAdd(errorX, errorXs),
64                    CTreeListCons(prunedX, prunedXs))
65      in
66        case curTree of
67          CLeaf(dist as CalculatedDist(mClass as Class Val4, scaledNumInstMajorityClass,
                scaledNumAtNode, scaledTotalN)) =>
68              (
69                case tor(rconst(0, 0.01, 0.0)) < scaledNumAtNode of
70                  true =>
71                    (case scaledNumAtNode * tor(rconst(0, 10.0, 100.0)) of
72                      numAtNode =>
73                        (numAtNode - (scaledNumInstMajorityClass * tor(rconst(0, 0.2, 2.0)
                              ) * numAtNode)) +
74                          errorEstimate(scaledNumInstMajorityClass, scaledNumAtNode,
                                scaledTotalN))
75                    | false => tor(rconst(0, 0.01, 0.0)),
76                    curTree
77              )
78          | CDN(splitPoint,
79                dist' as CalculatedDist( mClass' as Class Val4',
                      scaledNumInstMajorityClass', scaledNumAtNode', scaledTotalN' ),
80                children) => (
81                  case pruneCTreeList(children) of
82                    Pair1 as (childError, prunedChildren) =>
83                      case (scaledNumAtNode' * tor(rconst(0, 10.0, 100.0))) of
84                        numAtNode' =>
85                          case (numAtNode' - (scaledNumInstMajorityClass' * tor(rconst(0,
                                0.2, 2.0)) * numAtNode')) + errorEstimate(
                                scaledNumInstMajorityClass', scaledNumAtNode', scaledTotalN')
                                 of
86                            error =>
87                              case ((error - (childError + tor(rconst(0, 0.01, 0.1)))) <
                                    tor(rconst(0, 0.0000005, 0.000001))) of
88                                true =>
89                                  (error, CLeaf(dist'))
90                                | false =>
91                                  (childError, CDN(splitPoint, dist', prunedChildren))
92          )
93      end
94  end
95
96  fun main(curTree: c_tree): c_tree =
97    case f(curTree) of
98      (error, prunedTree) => prunedTree
99
100 %%
101 datatype data_set_info = DataSetInfo of int uncheckedArray * int
102 datatype instance = InstanceNil | InstanceCons of attribute_value * instance
103 datatype training_instance = TrainingInstance of (instance * class_value)
104 datatype data = DataNil | DataCons of training_instance * data
105 datatype distribution = DistributionNil | DistributionCons of real * distribution
106 datatype data_distribution = DataDistribution of data * distribution
107 datatype data_split = DataSplitNil | DataSplitCons of data_distribution * data_split
108 datatype split_point_list = SplitPointListNil | SplitPointListCons of split_point *
        split_point_list
109 datatype tree =
110   Leaf of distribution * class_value |
111   DN of split_point * distribution * tree_list
112 and tree_list = TreeListNil | TreeListCons of tree * tree_list
113
114 (* %{{{ Begin id3 *)
115 [snip...snip]
116 (* %}}} End id3 functions *)
117
118 (* %{{{ Begin util functions *)
119 [snip...snip]
120 (* %}}} end util functions *)
```

```
121 |
122 | (* %{{{ Begin conversion functions *)
123 | fun listToInstance nil = InstanceNil
124 | | listToInstance(x :: xs) = InstanceCons(x, listToInstance xs)
125 |
126 | local
127 |     fun toData convertToInstance data =
128 |     let
129 |
130 |         fun helper nil = DataNil
131 |         | helper((instance, classValue) :: xs) = DataCons(
132 |                 TrainingInstance(
133 |                    convertToInstance instance,
134 |                     classValue),
135 |                 helper(xs))
136 |     in
137 |         helper(data)
138 |     end
139 |
140 |     fun vectorToInstance( x: attribute_value vector) =
141 |       listToInstance (vectorToList x)
142 | in
143 |     val listAttributeVectorClassValueToData  = (toData vectorToInstance)
144 | end
145 |
146 | fun listToSplitPointList nil = SplitPointListNil
147 |     | listToSplitPointList(x :: xs) = SplitPointListCons(x, listToSplitPointList(xs))
148 |
149 | fun splitPointDecreaseIndex(sp) =
150 |     case sp of
151 |       NominalSplit(i, numAttributeValues) => NominalSplit(i -1, numAttributeValues)
152 |       | ContinuousSplit(i, v) => ContinuousSplit(i -1, v)
153 |
154 | fun splitPointsToDataSetInfo(splitPoints, numClasses) =
155 | let
156 |   val len = List.length(splitPoints)
157 |   fun helper(a, nil) = a
158 |   | helper(a, sp :: sps) =
159 |     case getAttributeIndex(sp) of
160 |       index =>
161 |         case index >= 0 andalso index < len of
162 |           true => (uncheckedArrayUpdate(a, index, getAttributeLength(sp));helper(a,sps))
163 |           | false => raise D1
164 |
165 | in
166 |   DataSetInfo(helper(uncheckedArray(len, ~1), splitPoints), numClasses)
167 | end
168 |
169 | fun createScaledCalculatedDist(cl, c, n, N) =
170 | if n <= 0.0 orelse N <= 0.0 then
171 |   CalculatedDist(cl, 0.0, 0.0, 0.0)
172 | else
173 |   let
174 |     val sc = c / ( n * 2.0)
175 |     val sn = n / 100.0
176 |     val sN = (5.0 * n) / N
177 |   in
178 |     CalculatedDist(cl, sc, sn, sN)
179 |   end
180 |
181 | fun scaleCalculatedDist(CalculatedDist(cl, c, n, N)) =
182 |   createScaledCalculatedDist(cl, c, n, N)
183 |
184 | fun scaleCTree(CLeaf(dist)) = CLeaf(scaleCalculatedDist(dist))
185 | | scaleCTree(CDN(sp, dist, children)) = CDN(sp, scaleCalculatedDist(dist),
        scaleCTreeList(children))
186 | and
187 | scaleCTreeList(CTreeListNil) = CTreeListNil
188 | | scaleCTreeList(CTreeListCons(x,xs)) = CTreeListCons(scaleCTree(x),scaleCTreeList xs)
189 |
190 | fun distToCalculatedDist(someClass, totalNum, dist) =
191 | let
192 |   exception CustomE of string
193 |
194 |   fun emptyDist( DistributionNil ) = true
195 |   | emptyDist(DistributionCons(x, xs)) = Real.==(x,0.0) andalso emptyDist(xs)
196 |
197 |   val majorityClass = majorityClass dist
198 |   val numMajority = numInstancesMajorityClass dist
199 |   val numAtNode = distributionSum dist
200 |   val RTotalNum = Real.fromInt(totalNum)
201 |
202 |   val empty = emptyDist dist
```

```
203 |   val majorityClass = (
204 |     case someClass of
205 |       SOME(actualMajorityClass) =>
206 |         if empty then
207 |           actualMajorityClass
208 |         else
209 |           if actualMajorityClass = majorityClass then
210 |             majorityClass
211 |           else
212 |             raise CustomE("Mismatch between the class of the node and the majorty Class
                      according to the distribution")
213 |       | NONE => if empty then raise CustomE("Should not be empty in this case") else
                majorityClass
214 |   )
215 | in
216 |   createScaledCalculatedDist(
217 |     majorityClass,
218 |       numMajority,
219 |       numAtNode,
220 |       RTotalNum)
221 | end
222 | fun treeToCTree (
223 |       trainSize,
224 |       Leaf(dist, classValue)) =
225 |   CLeaf(distToCalculatedDist(SOME classValue, trainSize, dist))
226 | | treeToCTree(
227 |     trainSize,
228 |     DN(
229 |       splitPoint,
230 |       dist,
231 |       children)) =
232 |         CDN(
233 |           splitPoint,
234 |           distToCalculatedDist(NONE, trainSize, dist),
235 |           treeListToCTreeList(trainSize, children)
236 |         )
237 | and treeListToCTreeList(trainSize, TreeListNil) = CTreeListNil
238 | | treeListToCTreeList(trainSize, TreeListCons(x, xs)) =
239 |     CTreeListCons(
240 |       treeToCTree(trainSize, x),
241 |       treeListToCTreeList(trainSize, xs)
242 |     )
243 | (* %}}} End conversion functions *)
244 |
245 | (* %{{{ Begin classification *)
246 | fun splitPointIndexInstanceVector((splitPoint, inst): split_point * attribute_value
      vector): int =
247 |   case splitPoint of
248 |     NominalSplit(index, len) => (
249 |       case Vector.sub(inst, index) of
250 |         Nominal(value) => value
251 |         | _ => raise D1
252 |     )
253 |     | ContinuousSplit(index, splitValue) => (
254 |       case Vector.sub(inst, index) of
255 |         Continuous(value) => if value < splitValue then 0 else 1
256 |         | _ => raise D1
257 |     )
258 |
259 | fun subTreeList((index, treeList):  int * tree_list): tree =
260 |     case treeList of
261 |         TreeListNil => raise D2
262 |         | TreeListCons(x, xs) =>
263 |             case 0 = index of
264 |                 true => x
265 |                 | false => subTreeList(index - 1, xs)
266 |
267 | fun classify((tr, inst): tree * attribute_value vector):class_value =
268 |   case tr of
269 |         Leaf(dist, classValue) => classValue
270 |         | DN(splitPoint, dist, treeList) =>
271 |           classify(
272 |             subTreeList(
273 |               splitPointIndexInstanceVector(splitPoint,  inst),
274 |               treeList),
275 |             inst
276 |           )
277 |
278 | fun subCTreeList((index, treeList):  int * c_tree_list): c_tree =
279 |     case treeList of
280 |         CTreeListNil => raise D2
281 |         | CTreeListCons(x, xs) =>
282 |             case 0 = index of
```

```
283                 true => x
284                 | false => subCTreeList(index - 1, xs)
285
286 fun classifyCTree((tr, inst): c_tree * attribute_value vector):class_value =
287   case tr of
288         CLeaf(CalculatedDist(classValue, _, _, _)) => classValue
289         | CDN(splitPoint, dist, treeList) =>
290             classifyCTree(
291                 subCTreeList(
292                   splitPointIndexInstanceVector(splitPoint,  inst),treeList
293                 ),
294                 inst
295             )
296 (* %}}} End classification *)
297
298 (* %{{{ Begin random *)
299 [snip..snip]
300 (* %}}} End random *)
301
302 fun sumCorrectAndWrong classify (tree, testInstances) =
303 let
304   fun addCorrectOrWrong((instance, classValue), (correct, wrong))=
305     if classify(tree, instance) = classValue then (correct+1, wrong) else (correct,
          wrong+1)
306 in
307   List.foldr addCorrectOrWrong (0,0) testInstances
308 end
309
310 (* %{{{ Begin synthetic data *)
311 exception CustomE of string
312
313 fun getTripplePermutations(oneList, twoList, threeList) =
314   List.concat(List.concat(
315     List.map
316       (fn one =>
317           List.map
318             (fn two =>
319               List.map
320                 (fn three =>
321                   (one, two, three)
322                 )
323               threeList
324             )
325           twoList
326         )
327         oneList
328   ))
329
330 fun shuffleArray(a) =
331 let
332   val len = Array.length a
333 in
334     Array.appi
335       (fn (index, _) =>
336       let
337
338         val rIndex = index + getRandomInt(len - index)
339         val v = Array.sub(a, index)
340         val rv = Array.sub(a, rIndex)
341         val _ = Array.update(a, index, rv)
342         val _ = Array.update(a, rIndex, v)
343       in
344         ()
345       end)
346     a
347 end
348
349 fun attributeIndexValuePermutations(postfix, splitPoints): (int * attribute_value) list
      list =
350 let
351   fun addPermutationsForSplitPoint(sp, l) =
352   let
353     val (attribIndex, attribLength) = (case sp of
354                                          NominalSplit(aIndex, aLength) => (aIndex,
                                               aLength)
355                                          | ContinuousSplit(aIndex, _) => (aIndex, 2)
356                                        )
357   in
358     List.concat(List.tabulate(attribLength,
359         fn attribValue => List.map (fn x => (attribIndex, Nominal(attribValue)) :: x) l)
              )
360   end
361 in
```

```
362   (List.foldr addPermutationsForSplitPoint [postfix] splitPoints)
363 end
364
365 fun permutations(splitPoints, handledSplitPoints, classValue, numAttributes): (
        attribute_value vector * class_value) list =
366 List.map
367   (fn permutation =>
368     let
369       val a = Array.array(numAttributes, Nominal(~1))
370       val _ = List.app (fn (ai, av) => Array.update(a, ai, av)) permutation
371     in
372       (Array.vector a, Class(classValue))
373     end
374   )
375   (attributeIndexValuePermutations(handledSplitPoints, splitPoints))
376
377 fun removeSplitPoint(splitPoints, splitPoint) =
378   List.filter (fn x => not( splitPointEq(splitPoint,x))) splitPoints
379
380 fun listToTrainTest (data, fraction, stratified) =
381 let
382   val _ = (fraction >= 0.0 andalso fraction <= 1.0)
383             orelse raise CustomE("fraction must be between 0 and 1.0")
384   val breakPoint = Real.floor(fraction * Real.fromInt(List.length data))
385 in
386   listFoldri
387     (fn (i, x, (ll, lr) ) => if i < breakPoint then (x :: ll, lr) else (ll, x:: lr))
388     ([],[])
389      data
390 end
391
392 fun createBinarySplitPoints(numAttributes) =
393     List.tabulate(numAttributes,
394       fn i => NominalSplit(i, 2)
395 )
396
397 fun intPermutations(lengths) =
398 let
399   fun addToPermutations(x, ll) =
400     List.concat(List.tabulate(x,
401         fn value => List.map (fn l => value :: l) ll))
402 in
403   (List.foldr addToPermutations [[]] lengths)
404 end
405
406 fun randomWeights(length) =
407   List.tabulate(length, fn x => getRandomReal() - 0.5)
408
409
410 fun maxIndex(values: (real) list) = let
411 val (index, value) =
412   listFoldri (fn (ix, vx, (iy, vy)) => if vx >= vy then (ix, vx) else (iy, vy)) (~1, ~
        Real.maxFinite) values
413
414 val _ = (index <> ~1) orelse raise CustomE("index is ~1, and this should not happen")
415 in
416   index
417 end
418
419 fun linearCombination attributes weights =
420 let
421  val _ = List.length(weights) = List.length(attributes) orelse raise CustomE("Wefihts
       and attributes must have the same length")
422
423 in
424   ListPair.foldr (fn (w, a, sum) => (w * a) + sum) 0.0 (weights, attributes)
425 end
426
427
428 fun splitPointsToLengthList(splitPoints) = List.map (fn x => getAttributeLength(x))
      splitPoints
429
430 fun addBiasToAttributes functions = List.map (fn weights => 1 :: weights) functions
431
432 fun multiLayerNetwork(weightsToHiddenNodes, weightsToOutputNodes:real list list,
      withBias) attributes =
433 let
434     val attributes = List.map Real.fromInt attributes
435     val hiddenValues = List.map (tanh o (linearCombination attributes))
          weightsToHiddenNodes
436     val hiddenValues = if withBias then 1.0 :: hiddenValues else hiddenValues
437
438
```

```
439     val _ = List.length(hiddenValues) = List.length(List.hd(weightsToOutputNodes))
            orelse raise CustomE("Mismath between hiddenValues and the weights of the output
            nodes")
440     val outputValues = List.map (linearCombination hiddenValues) weightsToOutputNodes
441 in
442   maxIndex(outputValues)
443 end
444
445 fun generateDataSetUsingEncoder encoder (splitPoints, network: (real list list *  real
        list list), withBias) =
446 let
447   val permutations = intPermutations( splitPointsToLengthList splitPoints )
448
449   val attributeValuesList = List.map (fn permutation => List.map (fn x => Nominal(x))
          permutation) permutations
450
451   val encodedPermutations = List.map encoder  permutations
452
453   val encodedPermutations = if withBias then addBiasToAttributes encodedPermutations
          else encodedPermutations
454
455   val (hw, ow) = network
456   val classes = List.map (multiLayerNetwork (hw,ow,withBias)) encodedPermutations
457
458   val numAttribs = List.length splitPoints
459   val _ = List.all (fn a => (List.length a) = numAttribs) attributeValuesList
460             orelse raise CustomE("wrong with generation of the data. Mismatch between
                  splitpoints and attributes")
461 in
462   ListPair.map (fn (a, v) => (Vector.fromList a, Class(v))) (attributeValuesList,
          classes)
463 end
464
465
466 val sumInt = (List.foldr (fn (r, y) => r + y ) 0)
467
468 fun binaryList value =
469 let
470     val _ = value >= 0 orelse raise CustomE("The value must be positive")
471
472     val next = value div 2
473     val rest = value mod 2
474 in
475     if next = 0 then
476         [rest]
477     else
478         (binaryList next) @ [rest]
479 end
480
481
482 fun getIndicatorWidths splitPoints =
483   List.map (fn l => List.length (binaryList (l - 1))) (splitPointsToLengthList
          splitPoints)
484
485
486 fun numIndicatorNodes(splitPoints) =
487   sumInt(
488     getIndicatorWidths splitPoints
489   )
490
491
492
493 fun indicatorEncoder splitPoints (attributeValues) =
494 let
495
496   val fixedWidths = getIndicatorWidths(splitPoints)
497
498
499
500   fun fixedWidthBinaryList(fixedWidth, value) =
501   let
502     val bns = binaryList(value)
503     val diff = fixedWidth - List.length(bns)
504     val _ = diff >= 0 orelse raise CustomE("The attribute value is too large when
            compared to the splitpoint")
505   in
506     (List.tabulate(diff, fn _ => 0)) @ bns
507   end
508
509 in
510   List.concat( ListPair.map fixedWidthBinaryList (fixedWidths, attributeValues) )
511 end
512
```

```
513 |
514 |
515 | fun perturb( W1 : real, W2 : real, Lower : real, Upper : real ) : real * real =
516 | let
517 |   val S = W1 + W2
518 |   val W =  Lower + randReal() * ( Upper - Lower )
519 |   val W' = S - W
520 |   val Eps = 1.0e~9
521 |   fun ok X = Lower - Eps <= X andalso X <= Upper + Eps
522 | in
523 |   if ok W andalso ok W' then
524 |     ( W, W' )
525 |   else
526 |     perturb( W1, W2, Lower, Upper )
527 | end (* fun perturb *)
528 |
529 |
530 | fun randomChanges( Xs : real list, Lower : real, Upper : real ) : real list =
531 | let
532 |   val A = Array.fromList Xs
533 |   fun randIndex() = randRange( 0, Array.length A - 1 )
534 |   fun g() =
535 |   let
536 |     val I = randIndex()
537 |     val K = randIndex()
538 |   in
539 |     if I = K then g() else
540 |   let
541 |     val ( Wi, Wk ) =
542 |       perturb( Array.sub( A, I ), Array.sub( A, K ), Lower, Upper )
543 |   in
544 |     Array.update( A, K, Wi );
545 |     Array.update( A, I, Wk )
546 |   end
547 |   end
548 | in
549 |   for( 1, Array.length A * 10, fn _ => g() );
550 |   array_to_list A
551 | end
552 |
553 |
554 |
555 | fun randomCustomWeights(length, from, to) =
556 |   List.tabulate(length, fn x => (((to - from) * getRandomReal())  + from))
557 |
558 | fun randomNetworks(numInputNodes, numHiddenNodes, numClasses, withBias) =
559 | let
560 |   val numInputNodes = if withBias then numInputNodes + 1 else numInputNodes
561 |   val Sigma = randReal() * 4.0
562 |   val Limit = Sigma * Math.sqrt 3.0 / Math.sqrt( real numInputNodes )
563 |   val HiddenWeights =
564 |     randomCustomWeights(numInputNodes, ~Limit, Limit)
565 |   val OutputWeights =
566 |     randomCustomWeights(
567 |       if withBias then numHiddenNodes + 1 else numHiddenNodes, ~1.0, 1.0)
568 | in
569 |   (
570 |     List.tabulate(numHiddenNodes,
571 |       fn x => randomChanges( HiddenWeights, ~Limit, Limit ) ),
572 |
573 |     List.tabulate(numClasses,
574 |       fn x =>  randomChanges( OutputWeights, ~1.0, 1.0 ) )
575 |   )
576 | end
577 |
578 | fun encodedLinesCreateInputOutput (numEncodedNodes, encoder) (splitPoints, numClasses,
      |  withBias)=
579 | let
580 |
581 |   val numNodes = numEncodedNodes(splitPoints)
582 |
583 |   val numHiddenNodes = randRange(5, 11)
584 |   fun g() =
585 |   let
586 |   val functions = randomNetworks(numNodes, numHiddenNodes, numClasses, withBias)
587 |
588 |   val data = generateDataSetUsingEncoder encoder (splitPoints, functions, withBias)
589 |   val Classes = map( fn( _, Class I ) => I, data )
590 |   val Histogram = Array.array( numClasses, 0 )
591 |   val () = loop( fn I =>
592 |     Array.update( Histogram, I, Array.sub( Histogram, I ) + 1 ),
593 |     Classes )
594 |   val Xs = map( real, array_to_list Histogram )
```

```
595   val X = min( op<, Xs )
596   in
597   if X < ( 1.0 / real numClasses / 5.0 ) * real_sum Xs then g() else (
598     p"\nHistogram_=_"; print_int_list( array_to_list Histogram );
599     ( functions, data ) )
600   end (* fun g() *)
601 in
602   g()
603 end
604
605 fun generalCreateInputOutput getModelAndData (splitPoints, numClasses, amountTraining,
        stratified)=
606 let
607   val (model, data) = getModelAndData(numClasses, splitPoints)
608   val a = Array.fromList data
609   val _ = shuffleArray a
610   val (train, test) = listToTrainTest(arrayToList a, amountTraining, stratified)
611   val dInfo = splitPointsToDataSetInfo(splitPoints, numClasses)
612 in
613   (model, dInfo, train, test)
614 end
615
616 fun removeSplitPointWithArranging(splitPoints: split_point list, index) =
617   listFoldri
618     (fn (i, x, y) =>
619       if i > index then
620         splitPointDecreaseIndex(x) :: y
621       else if i = index then y
622       else x :: y
623     )
624     [] splitPoints
625
626 fun removeRandomAttribute(splitPoints, data) =
627 let
628   val numAttributes = List.length splitPoints
629   val index = getRandomInt(numAttributes)
630   val splitPoints = removeSplitPointWithArranging(splitPoints, index)
631   val _ = (numAttributes = (List.length splitPoints + 1)) orelse
632             raise CustomE("Did_not_remove_attribute")
633   val data = List.map (fn (v,c) => (vectorRemoveIndex(v, index),c)) data
634
635 in
636   (splitPoints, data)
637 end
638
639 fun removeNRandomAttributes(splitPoints, data, n) =
640   List.foldr
641     (fn (_, (splitPoints, data)) =>
642
643       removeRandomAttribute(splitPoints, data)
644     )
645     (splitPoints, data)
646     (List.tabulate(n, fn x => x))
647
648 fun manipulateDataSetWithTrain(
649   (splitPoints,
650   data, numClasses),
651   (amountTraining,
652     amountOfTrainingToTraining,
653     numAttributesToRemove)) =
654 let
655     val testLimit = 3000
656     val (splitPoints, data) = removeNRandomAttributes(splitPoints, data,
          numAttributesToRemove)
657
658     val (train, test) = listToTrainTest(data, amountTraining, false)
659
660     val train = List.take(train, Real.floor(Real.fromInt(List.length train) *
          amountOfTrainingToTraining) )
661     val test = scramble test
662     val test = if List.length test > testLimit then List.take(test, testLimit) else test
663
664     val dInfo = splitPointsToDataSetInfo(splitPoints, numClasses)
665
666     val ttree = createTree(listToSplitPointList splitPoints, dInfo, (
          listAttributeVectorClassValueToData train))
667
668     val (trainC, trainW) = sumCorrectAndWrong classify(ttree, train)
669     val trainSize = List.length train
670     val (testC, testW) = sumCorrectAndWrong classify (ttree, test)
671     val testSize = List.length test
672     val input = ttree
673     val output = ((trainC, trainW, trainSize), (testC, testW, testSize), train, test)
```

```
674 in
675   (input, output)
676 end
677
678 fun manipulateDataSet(args) =
679   case manipulateDataSetWithTrain(args) of
680     (input, ((trainC, trainW, trainSize), (testC, testW, testSize), train, test)) =>
681       (input, ((trainC, trainW, trainSize), (testC, testW, testSize), test))
682
683 fun createAndManipulateDataset
684     createDataSet
685     manipulateDataSet
686     manipulationArgsList
687     createNewModelEachTime
688     numModels =
689       List.tabulate(numModels,
690         (fn x =>
691         let
692           val dataSetResult = createDataSet()
693         in
694           List.map
695             (fn manipulationArgs =>
696             let
697               val dataSetResult = if createNewModelEachTime then createDataSet() else
                      dataSetResult
698             in
699               manipulateDataSet(dataSetResult, manipulationArgs)
700             end)
701           manipulationArgsList
702         end
703         ))
704
705 fun createSynteticInputsOutputs
706       createDataSet
707       manipulateDataSet
708       modelArgsList
709       manipulationArgsList
710       numModels =
711   ListPair.unzip(List.concat(List.concat(
712     List.map
713       (fn modelArgs =>
714           (createAndManipulateDataset
715             (createDataSet modelArgs)
716             manipulateDataSet
717             manipulationArgsList
718             true(*remember to set this to true*)
719             numModels
720           )
721       )
722     modelArgsList )))
723
724 fun genDataFromFunctionModel(splitPoints, numClasses, _) =
725   #2(encodedLinesCreateInputOutput (numIndicatorNodes, indicatorEncoder splitPoints) (
        splitPoints, numClasses, true))
726
727 fun getRandomNumClasses maxNumClasses = (getRandomInt (maxNumClasses - 1)) + 2
728
729 fun genModel genData (splitPoints, maxNumClasses, depth) () =
730 let
731     val numClasses = getRandomNumClasses(maxNumClasses)
732     val (data) = genData(splitPoints, numClasses, depth)
733     val a = Array.fromList data
734     val _ = shuffleArray a
735 in
736   (splitPoints, arrayToList a, numClasses)
737 end
738
739 fun manipulateDataSetWithCTree(args) =
740   case manipulateDataSetWithTrain(args) of
741     (input, ((trainC, trainW, trainSize), (testC, testW, testSize), train, test)) =>
742       (treeToCTree(trainSize, input), ((trainC, trainW, trainSize), (testC, testW,
            testSize), test))
743
744 fun createRandomSplitPoints  (maxValue, numAttributes) =
745     List.tabulate(numAttributes,
746       fn i => NominalSplit(i, getRandomInt(maxValue-1) + 2)
747     )
748
749 fun getDataSetArguments() =
750 let
751   val splitPointsList = List.map (fn x => createRandomSplitPoints(4, x) )
752 [6, 7, 8, 9, 10]
753   val maxNumClassesList = [4]
```

```
754   val depthList = [~1]
755 in
756   getTripplePermutations(splitPointsList, maxNumClassesList, depthList)
757 end
758
759 fun getManipulationArguments() =
760 let
761
762   val amountTrainingList = [0.5]
763   val amountOfTrainingToTrainingList = [0.2, 0.25, 0.3, 0.35]
764
765   val numAttributeToRemoveList = rangeInt(0, 5, 1)
766 in
767   getTripplePermutations(amountTrainingList, amountOfTrainingToTrainingList,
          numAttributeToRemoveList)
768 end
769
770 val _ = MLton.Random.srand(Word.fromInt(1000))
771
772 val createInputsOutputs
773    = (createSynteticInputsOutputs
774       (genModel genDataFromFunctionModel)
775       (manipulateDataSetWithCTree)
776       (getDataSetArguments())
777       (getManipulationArguments())
778       )
779 (* %}}} End synthetic data *)
780
781 val numTrain = 1
782 val numTest = 10
783 val (Test_inputs, Test_outputs) = (createInputsOutputs numTest)
784 val (Validation_inputs, Validation_outputs) = (Test_inputs, Test_outputs)
785 val (Inputs, Outputs) = (createInputsOutputs numTrain)
786
787 val All_outputs =  Vector.fromList( Outputs @ Test_outputs )
788
789 val Funs_to_use = [
790   "false", "true", "=",
791   "realLess", "realAdd", "realSubtract", "realMultiply",
792   "realDivide", "tanh","tor", "rconstLess",
793   "CTreeListNil", "CTreeListCons",
794   "CLeaf", "CDN",
795   "CalculatedDist"
796   ]
797
798 val Reject_funs = []
799 fun restore_transform D = D
800
801 structure Grade : GRADE =
802 struct
803
804 type grade = unit
805 val zero = ()
806 val op+ = fn(_,_) => ()
807 val comparisons = [ fn _ => EQUAL ]
808 val toString = fn _ => ""
809 val fromString = fn _ => SOME()
810
811 val pack = fn _ => ""
812 val unpack = fn _ =>()
813
814 val post_process = fn _ => ()
815
816 val toRealOpt = NONE
817
818 end
819
820 val Abstract_types = [ "split_point" ]
821
822 val sumCAndW = sumCorrectAndWrong classifyCTree
823
824 fun output_eval_fun(I: int, _, prunedTree: c_tree) =
825 let
826   val ((treeTrainC, treeTrainW, treeTrainSize),
827        (treeTestC, treeTestW, treeTestSize),
828        testData) = Vector.sub(All_outputs, I)
829
830   val (prunedTreeC, prunedTreeW) = sumCAndW(prunedTree, testData)
831   val cImp = prunedTreeC - treeTestC
832   val c = Real.floor((Real.fromInt(cImp) / Real.fromInt(treeTestC)) * 10000.0)
833 in
834   {numCorrect = prunedTreeC, numWrong = prunedTreeW, grade = ()}
835 end
```

```
836 |   handle Ex =>
837 |     {numCorrect = 0, numWrong = 100000, grade = ()}
838 |
839 | fun calcDistEq(cd1 as CalculatedDist( C1, M1, N1, TN1 ), cd2 as CalculatedDist( C2, M2,
      N2, TN2 ) ) =
840 |       C1 = C2 andalso Real.==( M1, M2 ) andalso Real.==( N1, N2 ) andalso Real.==( TN1,
            TN2 )
841 |
842 | fun split_point_eq( NominalSplit X, NominalSplit Y ) = X = Y
843 | | split_point_eq( ContinuousSplit X, ContinuousSplit Y ) =
844 |       #1 X = #1 Y andalso Real.==( #2 X, #2 Y )
845 |   | split_point_eq( _, _ ) = false
846 |
847 | fun c_tree_eq( CLeaf( CalcDist1), CLeaf( CalcDist2) ) =
848 |       calcDistEq( CalcDist1, CalcDist2 )
849 |   | c_tree_eq( CDN( P1, Dist1, Xs1 ), CDN( P2, Dist2, Xs2 ) ) =
850 |   split_point_eq( P1, P2 ) andalso
851 |       calcDistEq( Dist1, Dist2 ) andalso
852 |       c_tree_list_eq( Xs1, Xs2 )
853 |   | c_tree_eq( _, _ ) = false
854 |
855 | and c_tree_list_eq( CTreeListNil, CTreeListNil ) = true
856 |   | c_tree_list_eq( CTreeListCons( X1, Xs1 ), CTreeListCons( Y1, Ys1 ) ) =
857 |       c_tree_eq( X1, Y1 ) andalso c_tree_list_eq( Xs1, Ys1 )
858 |   | c_tree_list_eq _ = false
859 |
860 | val Max_output_genus_card = 4
861 | val Max_output_genus_complexity = 1.2
862 |
863 | val Max_time_limit = 1000000
864 | val Time_limit_base = 1000000.0
865 | val Max_syntactic_complexity = 1.0e300
866 | val Use_test_data_for_max_syntactic_complexity = true
867 |
868 | val main_range_eq = c_tree_eq
869 |
870 | val Number_of_output_attributes = 1
```

Appendix G

Rewritten f Function for Pruning Specification 2

```
1  fun errorEstimate((sc, sn, sN): real * real * real): real =
2  let
3    val n = sn * 100.0
4    val e = n - ( sc * 2.0 * n)
5  in
6    if e < 1.0 then
7    let
8      val base = n * (1.0 - pow(0.25, 1.0 / n))
9    in
10     if 0.0 < e then
11       base + e * (
12         (errorEstimate((n-1.000000001)/(2.0 * n), sn, sN)
13       ) - base)
14     else
15       base
16   end
17   else
18     if n < e + 0.5 then
19       if 0.0 < n - e then n-e else 0.0
20     else
21     let
22       val errorRate = (e + 0.5) / n
23       val z = 0.674489751129221500
24
25       val sq = (errorRate / n) -
26             (errorRate * errorRate / n) +
27             (z * z / (4.0 * n * n))
28
29       val val1 = errorRate + (z * z) / (2.0 * n) + z * (sqrt sq)
30       val val2 = (1.0 + (z * z) / n)
31       val r = val1 / val2
32     in
33       ((r * n) - e)
34     end
35 end
36 and pruneCTreeList(CTreeListNil) = ( 0.0, CTreeListNil )
37 | pruneCTreeList( CTreeListCons(x, xs) ) =
38 let
39   val (errorX, prunedX) = f x
40   val (errorXs, prunedXs) = pruneCTreeList xs
41 in
42   ( errorX + errorXs, CTreeListCons(prunedX, prunedXs) )
43 end
44 and f( curTree as CLeaf( CalculatedDist(
45     class, sc, sn, sN) ) ) =
46 let
47  val n = sn * 100.0
48  val error = n - (sc * 2.0 * n)
49 in
50   (
51   if 0.0 < sn then
52     error + errorEstimate(sc, sn, sN)
53   else
```

```
54 |     0.0,
55 |   curTree
56 |   )
57 | end
58 | | f( CDN(splitPoint, dist, children) ) =
59 | let
60 |   val CalculatedDist( class, sc, sn, sN ) = dist
61 |   val (childError, prunedChildren) = pruneCTreeList(children)
62 |   val n = sn * 100.0
63 |   val error = (n - (sc * 2.0 * n)) + errorEstimate(sc, sn, sN)
64 | in
65 |   if error - (childError + 0.1) < 0.000001 then
66 |     (error, CLeaf dist)
67 |   else
68 |     (childError, CDN(splitPoint, dist, prunedChildren))
69 | end
```

Appendix H

Synthesized Program for Pruning Specification 2

```
fun f curTree =
  case curTree of
    CLeaf(
      dist as
        CalculatedDist(
          mClass as Class( Val4 ),
          scaledNumInstMajorityClass,
          scaledNumAtNode,
          scaledTotalN
          )
      ) => (
      case
        realMultiply(
          scaledNumAtNode,
          tor(
            rconst( 12, 0.32360293654397926, 0.6792176711838479E2 )
            )
          )       of
        numAtNode =>
          (
            realAdd(
              realSubtract(
                numAtNode,
                realMultiply(
                  realMultiply(
                    scaledNumInstMajorityClass,
                    tor(
                      rconst(
                        34,
                        0.8429684274190742E~4,
                        0.28584295366524395E1
                        )
                      )
                    ),
                  numAtNode
                  )
                ),
              scaledNumInstMajorityClass
              ),
            curTree
            )
      )
  | CDN(
      splitPoint,
      dist' as
        CalculatedDist(
          mClass' as Class( Val4' ),
          scaledNumInstMajorityClass',
          scaledNumAtNode',
          scaledTotalN'
          ),
      children
      ) =>
```

```
54  | case
55    realMultiply(
56      scaledNumAtNode',
57      tor(
58        rconst( 4, 0.10201505006249995E2, 0.5959799248750001E2 )
59        )
60      ) of
61    V62BA64 =>
62  case
63    let
64      fun g3CD3395 V3CD3396 =
65        case V3CD3396 of
66          CTreeListNil =>
67            (
68              tor( rconst( 0, 0.25, 0.2350543345568945 ) ),
69              CTreeListCons( curTree, CTreeListNil )
70              )
71        | CTreeListCons( V3CD3397, V3CD3398 ) =>
72        case f( V3CD3397 ) of
73          V3CD3399 as ( V3CD339A, V3CD339B ) =>
74        case g3CD3395( V3CD3398 ) of
75          V3CD339C as ( V3CD339D, V3CD339E ) =>
76            (
77              realAdd( V3CD339A, V3CD339D ),
78              CTreeListCons( V3CD339B, V3CD339E )
79              )
80    in
81      g3CD3395( children )
82    end of
83    V3CD339F as ( V3CD33A0, V3CD33A1 ) =>
84  case
85    realLess(
86      realSubtract(
87        realAdd(
88          realSubtract(
89            V62BA64,
90            realMultiply(
91              realMultiply(
92                scaledNumInstMajorityClass',
93                tor(
94                  rconst(
95                    9,
96                    0.25759437734844126E~1,
97                    0.2911831626001407E1
98                    )
99                  )
100               ),
101             V62BA64
102             )
103           ),
104         realAdd(
105           tanh(
106             realSubtract(
107               tanh(
108                 realAdd(
109                   tor(
110                     rconst( 3, 0.628125E~1, 0.1052447543004225E1 )
111                     ),
112                   V3CD33A0
113                   )
114                 ),
115               scaledTotalN'
116               )
117             ),
118           scaledTotalN'
119           )
120       ),
121       V3CD33A0
122       ),
123     scaledNumAtNode'
124     ) of
125   true => f( CLeaf( dist' ) )
126 | false => ( V3CD33A0, CDN( splitPoint, dist', V3CD33A1 ) )
```