Improving Random Forest Algorithm through Automatic Programming

Master's Thesis in Computer Science

Que Tran

May 15, 2015
Halden, Norway

Østfold University College www.hiof.no

Abstract

Random Forest is a successful ensemble prediction technique that exploits the power of many decision trees and judicious randomization to generate accurate predictive models. Recently, it has become one of the main current directions in Machine learning research. In this thesis, we aimed to investigate the possibility of improving the Random Forest using automatic programming, specially the Automatic Design of Algorithms Through Evolution (ADATE) system. To achieve the goal, we first studied the Random Forest algorithm from the perspective of a member in the family of ensemble learning methods. Based on this knowledge, we conducted two experiments using the ADATE system. In the first experiment, we attempted to improve the combination of base classifiers. The second experiment concentrated on improving the way in which base classifiers are generated. Although we did not succeed in our first experiment, the second experiment brought us good results. Experiments with 19 benchmark data sets showed that the best model we got achieves up to 14.3% improvement in performance (in total) compared with the original one.

Keywords: Random Forest, Ensemble Learning, ADATE, Automatic programming, Machine Learning

Acknowledgments

At the outset I would like to express my sincere gratitude to my supervisor, Assoc. Prof. Jan Roland Olsson, for his valuable suggestions, guidance, caring and encouraging support. Without him, my job would remain incomplete. I would like to thank Lars Vidar Magnusson, who was willing to help and gave me his best suggestions.

I would also like to thank my parents. They were always supporting and encouraging me with their best wishes.

Finally, I take this opportunity to thank all my friends, especially Hieu Huynh, who always stood by me, cheered me up and took me through loneliness.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Machine learning is the science of getting computers to adjust their actions so that they can act more accurately through examples. The field of machine learning, according to Michell [34], is concerned with the question of how to construct computer programs that automatically improve with experience. In the past decade, Machine learning has become so pervasive that people probably use it everyday without knowing. There have been many algorithms proposed, from the simplest ones, e.g. ZeroR, Linear Regression, to the much more complex algorithms, e.g. Deep Neural Networks or Support Vector Machine. However, it was shown experimentally that a given model based on an algorithm may outperform all others for a particular problem or for a specific data set, but it is abnormal to find a single model achieving the best results on the overall problem domain [20]. As a consequence, ensembles of models constitute one of the main current directions in Machine learning research.

In the family of ensemble learning methods, *Random Forest* is regarded as one of the most powerful one. It exploits the power of many decision trees, judicious randomization to generate accurate predictive models. Moreover, it also provides insights into variables importance, missing value imputations, etc. The random forest has remarkable few controls to learn, and therefore, analysts can effortlessly obtain effective models with almost no data preparation or modeling expertise. Besides, short training time and the ability to run in parallel are two other huge advantages of the random forest.

Since being introduced in 2001 by Breiman [12], the random forest method has attracted the attention of many researchers and practitioners. A number of ideas to improve the algorithm have been proposed. In this thesis, we focus on two main purposes. First, we aim at studying the random forest method. Instead of exploring the random forest as an isolated algorithm, we will first investigate the whole picture of the ensemble learning methods and then present the random forest algorithm as a member in that family. Although random forest is able to handle both classification and regression problems, within the scope of this thesis, we only focus on the classification ones. Improving the random forest algorithm used for regression problems may be considered in our future work. Our second purpose is to utilize the power of the Automatic Design of Algorithms Through Evolution (ADATE) system to improve the random forest algorithm.

1.2 Research question and method

Research question

As stated above, our primary target in this study is to improve the random forest method using the ADATE system. To achieve the goal, we need to carefully study the random forest algorithm in the perspective that it is a member in the ensemble learning methods family, thus having a general view and understanding the possibility of improving it. Moreover, using the ADATE system to improve an algorithm, especially a state-of-the-art one like random forest, is usually a time-consuming process. Typically, evolving such programs requires hundreds of millions of program evaluations. The need of choosing a small part of the algorithm, which can significantly improve the whole performance if it is improved, is therefore clear. Basically, at the end of this thesis, we need to answer the following research questions:

RQ *To what extent the Random Forest algorithm can be improved using the Automatic Design of Algorithms Through Evolution (ADATE) system?*
Secondary relevant research questions are:

- **RQ 1.1** *How can we implement the Random Forest algorithm correctly in Standard ML (SML)?*
- **RQ 1.2** *Which part of Random Forest is possible to be improved by the ADATE system?*
- **RQ 1.3** *Which extra-information do we need to prepare in advance to help the ADATE system to synthesize a solution effectively?*

Method

To improve the Random Forest using the ADATE system, we need to follow the following steps:

- *Implement the Random Forest algorithm in Standard ML* - As stated before, within the scope of this thesis, we will concentrated on improving classification Random Forest. In this algorithm, we use C4.5 as the base algorithm to develop decision tree classifiers. More information about C4.5 will be presented in Chapter 2.
- *Select the parts which will be improved* - Choosing a part of an algorithm that is most likely to be improved using the ADATE system or choosing the extra-information that should be prepared beforehand is a trial-and-error task. We need, first, to get a deep understanding of the algorithm, and then based on that knowledge, conduct experiments with different possible solutions to find out the best ones.
- *Write specification files* - Writing a specification file mainly involves the tasks of converting the parts, written in Standard ML, that will be evolved by the ADATE system into ADATE ML, choosing the suitable data sets and defining some necessary functions, such as fitness function and helping function.

1.3 Report Outline

The rest of the report is organized as follows.

- Chapter 2 introduces the *Decision Tree*, a predictive model from which a random forest model is constructed. In this chapter, we describe a basic decision tree algorithm as well as some attribute selection measures and pruning tree methods that are commonly used.
- In Chapter 3, we give an overview of *Ensemble learning methods* and introduce some strategies to construct a good ensemble, including making an ensemble diverse, combining classifiers and selecting ensemble size.
- *Random Forest* algorithm is described in Chapter 4. Besides, in this chapter, we also present some state-of-the-art works focusing on improving Random Forest in various manners.
- Chapter 5 is started by presenting the automatic programming, the basic of functional programming and ML language. In this chapter, we also give a brief introduction to ADATE system.
- In Chapter 6 we describe our experiments by showing how the experiments were designed and implemented.
- Chapter 7 shows the results of our experiments. We also explain in detail the differences between each improved program generated by the ADATE system and the original program.
- Finally, in Chapter 8, we conclude our work and draw the directions for future work.

Chapter 2

Decision Tree

A decision tree is a predictive model which can be used to approximate discrete-valued target functions. Decision tree are usually represented graphically as hierarchical structure. The topmost node, which does not have any incoming edge, is called *root node*. A node with outgoing edges are called *internal node*. Each internal node denotes a test on an attribute. Each edge represents an outcome of the test. All other nodes are *leaf nodes*. Each leaf holds a class label. When classifying a new instance, the instance is navigated from the root node down to the leaf, according to the outcome of the tests along the path. The class label in the leaf node indicates the class to which the instance should belong. A typical decision tree is shown in Figure 2.1. It represents the concept *buys˙computer*, that is, it predicts whether a customer is likely to purchase a computer based on the speed of the CPU, screen size and price.

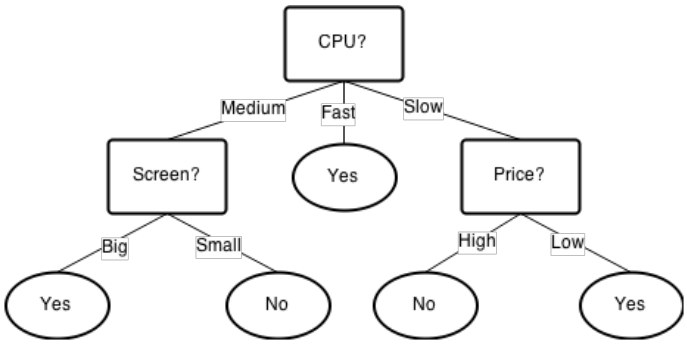

Figure 2.1: A decision tree for the concept *buys_computer*

Denote D as a data partition, *attribute_list* is a list of candidate attributes describing the data set and *Attribute_selection _method* is a heuristic method for selecting the splitting criterion that "best" separates a given data partition, D. A basic decision tree algorithm, called $buildTree(D, attribute\_list)$ is summarized as follows.

- The tree first starts as a single node N
- If the instances in D are all of the same class, N becomes a leaf and labeled with that class.

- Otherwise, *Attribute_selection_method* is called to decide the splitting criterion. The splitting criterion indicates the splitting attribute and may also indicate a split point. If the splitting attribute is nominal, it will be removed from the attribute list.
- The node N is labeled with the splitting criterion, which serves as a test at the node. A branch is grown from node N for each of the outcomes of the splitting criterion. The training instances in D are partitioned accordingly into, for example, $D_1, D_2, ...D_m$.
- Let D_i be the set of instances in D satisfying outcome i. If D_i is empty, N is attached a leaf labeled with the majority class in D. Otherwise, it is attached the node returned by $buildTree(D_i, attribute\_list)$. The recursive partitioning stops when any one of the following terminating conditions is reached.
  - All instances in the training set belong to a single class.
  - There are no remaining attributes which can be used for further partition.
  - There are no instances for a given branch.

  Besides three stopping criteria presented above, in some algorithms, there are some other conditions, such as the maximum tree depth has been reached, the number of cases in the terminal node is less than the minimum number of cases for parent nodes or the gained information at the best splitting criterion is not greater than a certain threshold.
- The resulting decision tree is returned

Decision tree learning is one of the most popular methods and has been successfully applied in many fields, such as finance, marketing, engineering and medicine. The reason for its popularity, according to many researchers, is that it is simple and transparent. The construction of a decision tree is fast and does not require any domain knowledge or parameter setting. Its representation in tree form is intuitive and easy to interpret for humans. However, successful use may depend on the data set at hand.

Many decision tree algorithms have been developed, including ID3 [42], C4.5(a successor of ID3) [41] and CART (Classification and Regression Trees) [10]. Most of them adopt a greedy approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Although those algorithms differ in many aspects, the main differences are their attribute selection measures and pruning tree methods. The next sections will present some attribute selection measures and pruning tree methods that are commonly used.

2.1 Attribute Selection Measures

2.1.1 Information gain

The ID3 algorithm [42] uses *information gain* as its *attribute selection measure*, which is a measure for selecting the splitting criterion that "best" separates a given data partition. The idea behind the method is to find which attribute would cause the biggest decrease in entropy if being chosen as a split point. The information gain is defined as the entropy of the whole set minus the entropy when a particular attribute is chosen.

The entropy of a data set is given by

$$Entropy(D) = -\sum_{i=1}^{m} p_i log_2(p_i) \tag{2.1}$$

where p_i is the probability that an instance in set D belongs to class C_i. It is calculated by $|C_i|/|D|$.

Suppose the attribute A is now considered to be the split point and A has v distinct values $\{a_1, a_2, ..., a_v\}$. Attribute A can be used to split D into v subsets $\{D_1, D_2, ...Dv\}$ where D_i consists of instances in D that have outcome a_j. The new entropy is defined by the following equation.

$$Entropy_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Entropy(D_j) \tag{2.2}$$

The information gain when using attribute A as a split point is as follows.

$$Gain(A) = Entropy(D) - Entropy_A(D) \tag{2.3}$$

$Gain(A)$ presents how much would be gained by branching on A. Therefore, the attribute A with the highest Gain(A) should be chosen to use.

2.1.2 Gain ratio

The information gain measure presented in section 2.1.1 is bias toward attributes having a large number of values, thus leading to a bias toward tests with many outcomes. C4.5 [41], a successor of ID3, uses an extension to information gain called *Gain ratio*, which attempts to overcome this shortcoming. The method normalize information gain by using a *split information* factor, defined as follows.

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times log_2(\frac{|D_j|}{|D|}) \tag{2.4}$$

Gain ratio is then given by the following equation.

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo(A)} \tag{2.5}$$

The attribute with the highest gain ratio is selected as the splitting point.

2.1.3 Gini index

The CART algorithm [10] uses the *gini index* as its attribute selection measure. The Gini index measures the impurity of set D. Therefore, it is also called *Gini impurity*. The Gini index only consider a binary split for each attribute. Gini index point of D is defined as follows.

$$Gini(D) = 1 - \sum_{i=1}^{m} {p_i}^2 \tag{2.6}$$

(The notation is the same as in the previous methods)

Suppose the attribute A is now considered to be the split point and A has 2 distinct values a_1, a_2. Attribute A can then be used to split D into D_1 and D_2 where D_i consists of instances in D that have outcome a_j. The gini index of D given that partitioning is given by the following equation.

$$Gini_A(D) = \frac{|D_1|}{|D|}Gini(D_1) + \frac{|D_2|}{|D|}Gini(D_2) \tag{2.7}$$

The reduction in impurity is defined as:

$$\Delta Gini(A) = Gini(D) - Gini_A(D). \tag{2.8}$$

The attribute with the highest reduction in impurity is selected for the next classification step.

2.1.4 ReliefF

Unlike the algorithms presented above, ReliefF [26] is not impurity based. It selects splitting points according to how well their values distinguish between similar instances. A good attribute is the one that can separate similar instances with different classes and leave similar instances with the same classes together.

Let D be the training set with n instances of p attributes. Each attribute is scaled to the interval [0, 1]. Let W be a p-long weight vector of zero. The algorithm will be repeated m times, and at each iteration, it chooses a random instance X. The closest same-class instance is called *near-hit*, and the closest different-class instance is called *near-miss*. The weight vector W is updated as follows.

$$W_i = W_{i-1} - (x_i - nearHit_i)^2 + (x_i - nearMiss_i)^2 \tag{2.9}$$

After m iterations, each element of the weight vector is divided by m. This vector is called *relevance vector*. Attributes are selected if their relevance is greater than a specified threshold.

2.2 Decision Tree Pruning

One challenge arising in a decision tree algorithm is to decide an optimal size of a tree. There is some stopping criteria proposed to control the size of a tree. However, employing tight stopping criteria tends to create a small tree which may not be able to capture important structural information in the training data. On the other hand, loose stopping criteria would lead to a large tree with a high risk of overfitting the training data. To tackle this problem, many *pruning methods* are presented. Pruning is a technique that reduces the size of decision trees by removing sections of the tree that do not contribute much in classifying instances. Researchers suggest using loose stopping criterion and allowing the decision tree to overfit the training set, then letting pruning methods to cut back the overfitted tree.

There are various techniques for pruning decision trees since it is one of the most extensively researched areas in machine learning. The following subsections will discuss the most popular pruning methods.

2.2.1 Reduced-Error Pruning

Reduced-Error Pruning, which as suggested by Quinlan [40], is one of the simplest strategy for simplifying trees. Starting with a complete tree, the algorithm tries to replace each node with the most frequent class ending at that node with respect to a test set. From all the nodes, the algorithm chooses the one at which the replacement makes the largest reduction in error rate to prune. The process is continued until there is no further pruning would increase or maintain the current accuracy.

This pruning method can end with the smallest accurate sub-tree with respect to a given test set.

2.2.2 Critical value pruning

Critical value pruning method was introduced by Mingers [33]. This is a bottom-up technique and similar to the reduced-error pruning method. However, instead of using the estimated error on test data to judge the quality of a sub-tree, this method relies on estimating the importance of a node from calculations done in the tree creation step.

As mentioned earlier in the *Introduction*, a decision tree algorithm recursively use a selection criterion to split the training data into smaller and purer subsets. At each node, the splitting point is chosen in the manner that maximizes the value of the splitting criterion. This value is also employed in the critical value pruning method to make pruning decisions. The value of the splitting criterion at a node is compared to a fixed threshold to decide if the node needs to be pruned. If the value is smaller than the threshold, the node will be pruned and replaced by a leaf. However, there is one more rule, which is if the sub-tree contains at least one node whose value is greater than the threshold, it will be kept. In other words, a sub-tree is only considered for pruning if all its successors are leaf nodes.

2.2.3 Cost-Complexity Pruning

Cost-complexity pruning was introduced in the classic CART system [10] for inducing decision trees. The method consists of two phases. In the first phase, a sequence of increasingly smaller pruned trees $T_0, T_1, ..., T_k$ is built, where T_0 and T_k are the original tree and the root tree respectively. Given a tree T_i, the successor tree T_{i+1} is obtained by replacing one or more of the sub-trees in T_i with suitable leaves. The pruned subtrees are those that lead to the smallest increase in error rate per pruned leaf. The increase in error is measured by a quantity α that is defined to be the average increase in error per leaf of the subtree.

$$\alpha = \frac{\varepsilon(pruned(T,t),S) - \varepsilon(T,S)}{|leaves(T)| - |leaves(pruned(T,t))|} \tag{2.10}$$

where $\varepsilon(T,S)$ is the error rate of the tree T over the sample S, $|leaves(T)|$ is the number of leaves in T and $pruned(T,t)$ denotes the tree obtained by replacing the node t in T with a suitable leaf.

After building a sequence of trees, in the next phase, based on the size of the given data set, CART either uses a hold-out set or cross-validation to estimate the error rate of each pruned tree. The best pruned tree is then selected.

2.2.4 Minimum-Error Pruning

The method was developed by Niblett and Bratko [35] with the idea behind is to compare the error rate estimation, at each node, with and without pruning.

If an internal node is pruned, it becomes a leaf, and its error rate is calculated by:

$$\varepsilon' = 1 - \max_{c_i \in dom(y)} \frac{|\sigma_{y=c_i} S_t| + l.p_{apr}(y = c_i)}{|S_t| + l} \tag{2.11}$$

where S_t denotes the instances that have reached a leaf t, $p_{apr}(y = c_i)$ is the *a-priori* probability of y getting the value c_i, and l is the weight given to the *a-priori* probability.

The expected error rate if the node is not pruned is calculated using the error rates for each branch, combined by weighting according to the proportion of observations along each branch. The procedure is performed recursively because the error rate for a branch cannot be calculated until we know if the branch itself is to be pruned. Finally, the error rate estimation for a certain internal node before and after pruning is compared. If pruning the node leads to a lower error rate, then the sub-tree is pruned; otherwise, it is kept.

The advantage of this method is that it minimizes the total expected error and does not require a separate test set. However, there are some drawbacks. First, it has an assumption of equally likely classes, which is seldom true in practice. Second, in this method, the pruning is strongly affected by the number of classes, thus leading to unstable results.

2.2.5 Pessimistic Error Pruning

This method was proposed by Quinlan [40] which aims to avoid the need of a test set or cross validation. The motivation for the method is that the mis-classification rates produced by a tree on its training data are overly optimistic. Therefore, Quinlan suggested using a more realistic measure, known as the continuity correction for the binomial distribution.

Let $N(t)$ denotes the number of training instances at node t, $e(t)$ denotes the number of instances mis-classified at node t. Then, an estimate of the mis-classification rate is:

$$r(t) = \frac{e(t)}{N(t)}. \tag{2.12}$$

and the rate with the continuity correction is:

$$r'(t) = \frac{e(t) + 1/2}{N(t)}. \tag{2.13}$$

For a sub-tree T_t the mis-classification rate will be

$$r(T_t) = \frac{\sum e(i)}{\sum N(i)}, \tag{2.14}$$

where i covers the leaves of the sub-tree. Thus the corrected mis-classification rate will be calculated by:

$$r'(T_t) = \frac{\sum (e(t) + 1/2)}{\sum N(t)} = \frac{\sum e(i) + N_T/2}{\sum N(i)}, \tag{2.15}$$

where N_T is the number of leaves.

However, this correction still produces an optimistic error rate. Hence, Quinlan suggested only keeping the sub-tree if its corrected number of mis-classifications is lower than that for the node by at least one standard error. The standard error for the number of mis-classification is defined as:

$$SE(n'(T_t)) = \sqrt{\frac{n'(T_t) \times (N(t) - n'(T_t))}{N(t)}} \tag{2.16}$$

where $n'(T_t) = \sum e(i) + N_T/2$.

Chapter 3

Ensemble Learning

Ensemble learning is a family of methods which generate multiple classifiers from the original data set and then try to combine them to constitute a new classifier which can obtain better performance than any of its constituents. Constructing good ensembles has become one of the most active areas of research in supervised learning because both empirical studies and specific machine learning applications verify that a given classification method outperforms all others for a particular problem or for a specific subset of the input data, but it is abnormal to find a single method achieving the best results on the overall problem domain [20]. Therefore, combining multiple learners to exploit the different behavior of the base classifiers to improve the accuracy has become a concern of many researchers and practitioners. There are hopes that if a single classifier fails, a committee of many classifiers can recover the error.

A typical ensemble framework usually contains the following components:

- **Training set generator**: The generator is responsible for creating training sets for all component classifiers of an ensemble. It is common that component classifiers are built from various training sets to make them act differently. However, in some algorithms, all classifiers are trained from the same data set and, in this case, making classifiers diverse is then the responsibility of inducers. The training set generator, in this situation, just needs to return the original data set for all classifiers.

- **Inducers**: The inducer is an algorithm that gets a training set and build a classifier that represents the relationship between the input attributes and the target attribute. All classifiers can be constituted from the same inducer or from many different inducers.

- **Combiner**: The role of a combiner is to combine the outputs from component classifiers to give a final prediction. There are various combiners, from simple ones to complicated ones. For example, one very simple way to combine the results of a classification problem is to use majority voting. For regression problems, rather than taking the majority vote, it is common to take the mean of the outputs.

Two families of the ensemble methods are usually distinguished based on the classifier dependency. They are dependent methods and independent methods.

In dependent approaches, the outcomes of a certain classifier affect the creation of the next classifier. In some algorithms, the classifiers constructed in the previous iterations are employed to manipulate the training set for the next iteration. These approaches usually

let the classifiers learn only from instances that are mis-classified by previous classifiers and ignore all other instances. Such method is called *Model-guided Instance Selection* [45]. One typical example of this method is AdaBoost algorithm formulated by Yoav Freund and Robert Schapire [22]. Another approach in this family of ensemble methods is *Incremental Batch Learning*. The approach uses the current training set together with the classification of the former classifier for building the next classifier. At the last iteration, the final classifier is constructed.

Contrary to dependent methods, in independent methods, each classifier is built independently. Their outputs are then combined in some fashion. Diversity of classifiers are gained by manipulating the training set or the classifiers. Some of the most well-known independent methods are Bagging [11], Random Forest [12], and Wagging algorithm [4].

The following sections will introduce some strategies to construct a good ensemble, including making an ensemble diverse, combining classifiers and selecting ensemble size.

3.1 Ensemble diversity

According to Krogh and Vedelsby (1995) [27], diversity of classifiers in an ensemble theoretically plays an important role in obtaining a good performance of the ensemble. Researchers have been introduced many approaches to create classifiers which are as different as possible while still have high accuracy. In the book named *Data mining with Decision Tree* [45], Lior Rokach and Oded Mainmon proposed the following taxonomy of those approaches.

- Manipulating the Inducer - In this method, the ways in which the inducer are used to generate classifiers are manipulated, thus creating different classifiers.
- Manipulating the Training Sample - The training set for each ensemble member is manipulated. In other words, each classifier is trained with a different training set.
- Changing the target attribute representation - Each classifier in an ensemble is assigned a task and solves a different target concept.
- Partitioning the search space - In this method, many search subspaces are created and each classifier is trained on one of those subspaces.
- Hybridization - An ensemble consists of various base classifiers.

Our report only presents the first two methods mentioned above because they are not only related to our main focus, which is Random Forest algorithm, but also are the most well-known methods that are frequently used in many ensembles.

3.1.1 Manipulating the Training Samples

In this method, each ensemble member is trained on a different subset of the original training set. According to [45], classifiers, such as decision tree and neural network, whose variance-error factor is relatively large may get a huge change even though there are small changes in the training set. Therefore, this method is suitable for such kinds of classifiers.

Resampling

In this approach, a new training set is created by taking instances from the original training set. One sample of resampling is *bootstrap sampling* method. Bootstrap sample is a new sample taken from an original data set with replacement. It is the same size as the original one. Hence, in the bootstrap sample, some data may appear several times and others not at all. Bootstrap sampling is used in several algorithms, such as Bagging and Random Forest. Instead of taking instances with replacement, some algorithms like AdaBoost and Wagging assign a weight to each instances in the training set. Classifiers then take those weights into account to create different ensemble members. The distribution of training instances in the new set can be random as in Bootstrap or approximately the same as that in the original set. [16] has shown that proportional distribution as used in combiner tree can achieve higher accuracy than random distribution.

DECORATE algorithm

The *DECORATE algorithm* (Listing 3.1) was proposed by Melville and Mooney (2003) [31]. The method was designed to use additional artificially generated training data to generate diverse ensembles. An ensemble is generated iteratively, one new classifier generated in each iteration is added into the current ensemble. At first step, an ensemble member is built using the base classifier on the original training set. In each iteration, a number of artificial training instances are generated based on a simple model of data distribution and then added into the training set. Labels for those artificial training instances are chosen so as to differ maximally from the current ensemble's predictions. The successive ensemble member is built on the new training set. Experiments have shown that this technique can achieve higher accuracy than boosting on small training sets and comparable performance on larger training sets.

```
1
2  Given:
3  T    set of training examples
4  U    set of unlabeled training examples
5  BaseLearn    base learning algorithm
6  k    number of selective sampling iterations
7  m    size of each sample
8
9    1. Repeat k times
10   2. Generate a committee of classifiers,
11        C* = EnsembleMethod(BaseLearn, T)
12   3. ∀x_j ∈ U, compute Utility(C*, x_j), based on the current committee
13   4. Select a subset S of m examples that maximizes utility
14   5. Label examples in S
15   6. Remove examples in S from U and add to T
16   7. Return EnsembleMethod(BaseLearn, T)
```

Listing 3.1: DECORATE algorithm

Partitioning

Handling massive data raises a challenge in loading the entire data set into a memory of a single computer. Chawla et al. (2004) [17] claimed that distributed data mining can address, to a large extent, the scalability and efficiency issues presented by massive

training sets. The data sets can be randomly partitioned into disjoint partitions with a size that can be efficiently managed on a group of processors. Each classifier is built on a disjoint partition and then can be aggregated. This is not only resolve the issue of memory but also leads to creating a diverse and accurate ensemble. In [17], Chawla et al. also proposed a framework for building thousand of classifiers that are trained from small subsets of data in a distributed environment. Empirical experiments have shown that the framework is fast, accurate and scalable. The performance of this approach is equivalent to the performance obtained by bagging.

3.1.2 Manipulating the Classifiers

To gain classifiers diversity, a simple and natural method is to manipulate the original classifier. There are several ways to do this.

Manipulation of the classifier's parameters

The original classifier can be modified by altering its parameters. Some changes in parameters can greatly affect the performance of the classifier. For instance, in decision tree classifier C4.5, the minimal number of instances per leaf, the confidence factor used for pruning and whether counts at leaves are smoothed based on Laplace are some of parameters that could be controlled to gain diversity.

In neural network classifier, networks can be made to be different by changing number of nodes, architecture, training algorithm or activation function.

Starting point in hypothesis space

Another method to gain diversity is to start the search in the hypothesis space in different points. For example the simplest way to manipulate the back-propagation inducer is to assign different initial weights to the network [38]. Empirical studies show that the number of cycles in which networks take to converge upon a solution, and in whether they converged at all can differentiate the results.

Traversing hypothesis space

Classifiers diversity is gained by altering the way in which the classifiers traverse the hypothesis space. One method is to inject randomness into the classifiers. For example, Ali and Pazzani [2] proposed that instead of selecting the best literal at each stage, the literal is selected randomly such that its probability of being selected is proportional to its measured value. There are also some other ways to inject randomness, such as randomly choosing a subset of attributes and then finding out the best among them in Random Forest algorithm [12], or randomly select an attribute from the set of the best 20 attributes in [20].

Another method to make an ensemble diverse was presented by Liu and Yao [28], namely *negative correlation learning*. In negative correlation learning, all the individual networks in the ensemble are trained simultaneously and interactively through the correlation penalty terms in their error functions. Rather than producing unbiased individual networks whose errors are uncorrelated, negative correlation learning can create negatively correlated networks to encourage specialization and cooperation among the individual networks [28]. The central idea behind the method is to encourage different

individual networks in an ensemble to represent different subspaces of the problem so that the ensemble can handle the whole problem better.

3.2 Combination methods

The following sections will focus on combination methods that are used to combine base classifiers in ensemble learning. There are two main methods: *weighting methods* and *meta-learning methods*. While weighting methods are usually used to combine classifiers built from a single learning algorithm, meta-learning is a good choice for combining classifiers from various learning algorithms.

3.2.1 Weighting methods

To combine classifiers with weighting methods, each classifier is assigned with a weight proportional to its strength. The weights can be static or dynamically change based on the instance to be classified. Some of the most well-known weighting methods are Majority voting, Performance weighting, Demster-Shafer method, Vogging, and mixture of experts.

Majority voting

In this method, an unlabeled instance is classified by all the classifiers. Each classifier votes for a class that the instance should belong to. The class with the most frequent vote will be assigned to the new instance. Therefore, this method sometimes is called *the plurality vote*.

Mathematically, the algorithm can be written as:

$$class(x) = argmax_{c_i \in dom(y)} \left(\sum_k g(y_k(x), c_i) \right) \tag{3.1}$$

where $y_k(x)$ is the classification of the k'th classifier and g(y, c) is an indicator function defined as:

$$g(y, c) = \begin{cases} 1 & y = c \\ 0 & y \neq c. \end{cases} \tag{3.2}$$

Performance weighting

Performance weighting method assigns each classifier a weight which is proportional to its accuracy performance on a validation set. [37]. The weight is defined as:

$$\alpha_i = \frac{1 - E_i}{\sum_{j=1}^{T} (1 - E_i)} \tag{3.3}$$

where E_i is a factor based on the performance of classifier ith on a validation set.

Demster-Shafer method

Shilen et al. [46] suggested a method for combining base classifiers which borrowed the idea from the Dempster-Shafer theory of evidence [13]. The method chooses the class that maximizes the value of the belief function:

$$Bel(c_i, x) = \frac{1}{A}.\frac{bpa(c_i, x)}{1 - bpa(c_i, x)} \tag{3.4}$$

where $bpa(c_i, x)$ is defined as follows.

$$bpa(c_i, x) = 1 - \prod_k (1 - \hat{P_{M_k}}(y = c_i|x)) \tag{3.5}$$

where $\hat{P_{M_k}}(y = c_i|x)$ is the probability assignment defined for a certain class c_i given the instance x. And

$$A = \sum_{\forall c_i \in dom(y)} \frac{bpa(c_i, x)}{1 - bpa(c_i, x)} + 1. \tag{3.6}$$

Vogging

Derbeko et al. [19] proposed an approach for aggregating an ensemble of bootstrapped classifiers. The new technique is called *Variance Optimized Bagging* or *Vogging*. The central idea behind the approach is to find a linear combination of base classifiers so that the weights are optimized to reduce variance while preserve a prescribed accuracy.

This technique was inspired by a theory from Mathematical Fiance called Markowitz Mean-Variance Portfolio Optimization. Suppose there are m assets S_1, S_2, ... S_m, denote r_i as a predicted expected monetary return for S_i, σ_i as a predicted standard deviation of the return of S_i and Q as the m $\times$ m covariance matrix. A portfolio is a linear combination of assets and it is expected to return $\sum_i^m w_i r_i$ where $w_i \in (w_1, w_2, ...w_m$ with $\sum_i^m w_i = 1$. The variance of a portfolio is used to measured its risk,

$$\sigma^2(w) = \sum_{i,j} w_i w_j Q_{ij} = w^t Q w \tag{3.7}$$

The output of the Markowitz algorithm is a *efficient frontier*. It is a set of portfolios with the highest expected return among those with the same or lesser risk, and the least risk among those with the same or greater return. Investors who want to find an "optimal" portfolio should choose a point on the efficient frontier curve. However, which exact portfolio will be chosen depends on personal utility functions of investors.

Applying the Markowitz algorithm to Machine learning, Derbeko et al. proposed the Vogging method as follows. [19]

Input

1. T (number of bagged classifiers)
2. k (number of efficient frontier points)
3. $S = (x_1, y_1), ..., (x_n, y_n)$ (training set)

4. H (base classifier hypothesis class)

Training

1. Generate T bootstrap samples, B_1, ... B_T from S
2. Train T classifiers h_i, ..., h_T such that $h_j \in H$ is trained over B_j
3. Let $\bar{A_j} = \frac{1}{T-1}\sum_{i \neq j} A_j(B_i)$ be the average empirical accuracy over all the other bootstrap samples. Evaluate $\bar{A_j}$, for all $j = 1, 2, ... T$ and Q.
4. Choose k uniformly spread points $a_1, ...a_k$ in $[min_j \bar{A_j}, max_j \bar{A_j}]$
5. The following quadratic program (QP) should be solved with linear constraints:

$$minimize(overw) : \frac{1}{2} w^t Q w \tag{3.8}$$

$$subjectto : (\bar{A_1}, ...\bar{A_T})^t, w \geq a \tag{3.9}$$

$$\sum_j w_j = 1, w \geq 0 \tag{3.10}$$

   Solve k instances of QP with the accuracy constraints $a_1, ...a_k$. For $i = 1, ...k$ let w_i and $(a_i^{'}, \sigma_i)$ be the result weight vector and mean-variance pair corresponding to a_i

6. Let p_0 be the proportion of the larger class in S

Output "Vogging weight vector" w_i^* with $i^* = argmax_i \frac{a_i^{'} - p_0}{\sigma_i}$

Mixture of experts

Mixture of expert (ME) is one of the most well-known combination methods. The ME method is proposed based on Divide-and-Conquer principle [25], in which the problem space is partitioned stochastically into a number of subspaces through special employed error function, experts become specialized on each subspace. A gating network is trained together with the experts, and is used to assign weights to the experts. Unlike other combination methods, instead of assigning a set of fixed combinational weights to the expert, the gating network compute these weight dynamically from the inputs, according to local efficiency of each expert.

Various ME strategies were presented to divide the problem spaces between the experts recently. Those implementations were classified into two groups based on the partitioning strategies used and both how and when the gating network is involved in the partitioning and combining procedures [30]. The first group, namely the Mixture of Implicitly Localised Experts (MILE), consists of the conventional ME and the extensions of this method that stochastically partition the problem space into a number of subspaces using a special error function. In the second group, the problem space is first partitioned by a clustering method and each expert is then assigned to one of these subspaces. This group is called Mixture of explicitly localised experts (MELE).

3.2.2 Meta-learning methods

Meta-learning is a method which learns from new training data created from the classifications of the base classifiers and some characteristics of them. Some of the most well-known meta-learning methods are Stacking, Arbiter Trees, and Combiner Trees.

Stacking

Stacking is a technique for achieving the highest generalization accuracy [49]. The central idea behind Stacking is to build a meta-dataset from the original dataset, then learn from it to form a final meta-classifier. The new meta-dataset is built using the predicted classifications of the base classifiers as input attributes. The target attribute remains as in the original dataset. In order to build the meta-dataset, the original dataset is usually divided into two subsets. The first one used to build base classifiers, and the second one is fed into those classifiers to form the meta-dataset.

The choice of input attributes and the learning algorithm at the meta-level are two most essential problems in Stacking. Much research has been proposed to address those issues. One of them is from Ting and Witten (1999) [47]. They suggested using Stacking with probability distributions and multi-response linear regression. It means instead of using only classifications from base classifiers, they add the probability distributions (PDs) over the set of class values. The reason for the extension, according to the authors, is that it would allows the meta-level classifier not to use only the predictions, but also the confidence of the base classifiers. Besides, multi-response linear regression (MLR) is recommended for meta-level learning. Suppose there are m class values, m regression problems are formulated: for each class c_j , a linear equation LR_j is constructed to predict a binary variable, which has value one if the class value is c_j and zero otherwise. Given a new example x to classify, LR_j(x) is calculated for all j, and the class k is predicted with maximum $LR_k(x)$ [47].

Another solution for the two issues in Stacking was recommended by Merz (1999) [32], called SCANN. The main idea behind the method is based on the knowledge that the more diverse base classifiers are, the better performance they yield. The correlations between the predictions of base classifiers are detected by SCANN using correspondence analysis. Then, meta-dataset is transformed to remove these correlations. In addition, SCANN employs the *nearest neighbor* method as its learning algorithm for meta-level learning step.

Arbiter Trees

The *arbiter tree* which is built in a bottom-up fashion was proposed by Chan and Stolfo (1993) [16]. In this method, the original data is partitioned randomly into many disjoint subsets from which classifiers are learned. An arbiter is built from the output of a pair of learned classifiers and recursively, an arbiter is learned from the output of two arbiters. An arbiter tree is generated with the initially learned base classifiers at the leaves. Therefore, for k subsets, there are k classifiers and $log_2(k)$ levels generated. A sample arbiter tree built from 4 base classifiers is shown in Figure 3.1 [16].

In detail, for each pair of classifiers, firstly, a validation set is formed by combining the data subsets on which the classifiers are trained. The validation set is then classified by the two classifiers. A selection rule compares the outputs from the two classifiers and selects instances from the validation set to form the training set for the arbiter. Same learning algorithm is used to build the arbiter. The process of forming the union of data subsets, classifying it using a pair of arbiter trees, comparing the predictions, forming a training set, and training the arbiter is recursively performed until the root arbiter is formed. The purpose of the selection rule is to choose examples that are confusing; i.e., the majority of classifiers do not agree [15]. There are three version of selection rules:

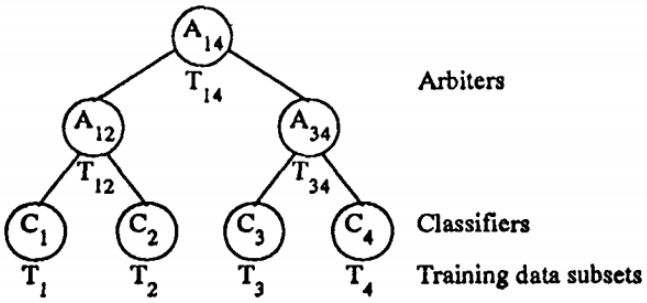

Figure 3.1: A sample Arbiter Tree

1. Return instances that are differently classified by classifiers, i.e. $T = T_d = \{x \in E | AT_1(x) \neq AT_2(x)\}$, in which $AT_i(x)$ denotes the prediction of training example x by arbiter subtree AT_i
2. Return instances in T_d, but also instances that incorrectly classified, i.e. $T_i = \{x \in E | (AT_1(x) = AT_2(x) \bigwedge class(x) \neq AT_i(x))\}$
3. Return a set of three training sets: T_d, T_i and T_c, where $T_c = \{x \in E | (AT_1(x) = AT_2(x) \bigwedge class(x) = AT_i(x))\}$ and $class(x)$ denotes the given classification of example x.

When an instance is classified by the arbiter tree, predictions flow from the leaves to the root. First, for each pair of classifiers, the predictions of the two classifiers and the parent arbiters prediction decide a final classification outcome based on an arbitration rule. This process is applied at each level until a final prediction is produced at the root of the tree. There are also two versions of arbitration rules. The task of determining which arbitration rule is utilized depends on the version of selection rule used for generating the training data at that level.

Combiner

The *Combiner strategy* is a meta-learning technique proposed by Chan and Stolfo [14]. The purpose of this method is to coalesce the predictions from the base classifiers by learning the relationship between these predictions and correct predictions. In this method, the training set for the combiner is formed from the outputs of base classifiers with the guidance of a composition rule. From those training examples, the combiner is built. When classifying a new instance, the base classifiers first generate their predictions. With those predictions, the instance is transformed to a new one by applying the same composition rule. The classification of the new instance using the combiner is then labeled for the investigated instance. Figure 3.2 [14] demonstrates a sample classification in the combiner strategy with 3 base classifiers.

There are three schemes for the composition rule:

1. Return meta-level training instance with the expected classification and the predictions of base classifiers, i.e., $T = \{class(x), C_1(x), C_2(x), ...C_k(x) | x \in E\}$, where $C_i(x)$ denotes the prediction of classifier C_i, $class(x)$ denotes the correct classification of example x as specified in the training set, E. This scheme is called *meta-class*.

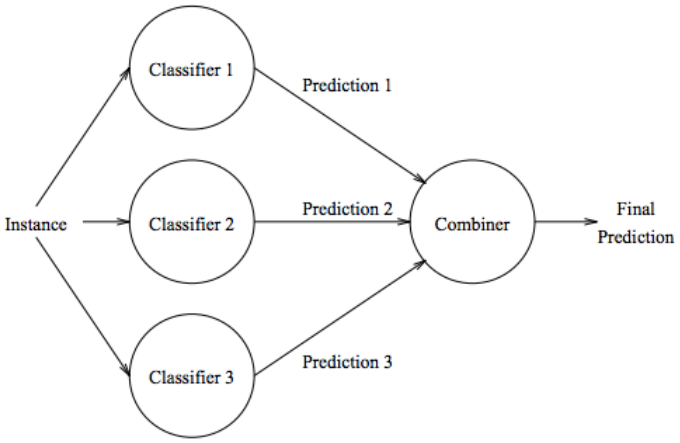

Figure 3.2: Classification in the combiner strategy

2. Return meta-level training instances similar to those in the meta-class scheme, plus the original attribute vectors of the training set of base classifiers. That means, $T = \{class(x),\ C_1(x),\ C_2(x), ...\ C_k(x),\ attrvec(x) \mid x \in E\}$. This scheme is called *meta-class-attribute*.

3. Return meta-level training instances similar to those in the meta-class scheme. However, each prediction C_i is replaced by m binary predictions $C_{i_1}(x), C_{i_2}(x), ..., C_{i_m}(x)$, where m is the number of classes. A binary classifier C_{i_j} is trained on instances which are labeled with class j and $\neg j$. In other words, this means, $T = \{class(x), C_{1_1}(x), C_{1_2}(x), ..., C_{1_m}(x), C_{2_1}(x), C_{2_2}(x), ..., C_{2_m}(x), ...C_{k_1}(x), C_{k_2}(x), ..., C_{k_m}(x) | x \in E\}$.

3.3 Ensemble size

Besides learning and combination methods, ensemble size, which is related to how many component classifiers should be used, also plays an important role in generating a good ensemble. According to [45], there are several aspects that may affect the size of an ensemble:

- **Accuracy**. Accuracy is usually known as the first priority factor that decides ensemble size. In most cases, ensembles containing ten classifiers are sufficient for reducing the error rate [24]. Regrading to the selection of decision trees in random forests, Bernard et al. [6] empirically showed that the error rate, in most datasets, drops significantly when the number of trees is around 10-15. When the number of trees grows a bit higher, the error rate reduces slightly. However, when the ensemble size exceeds a specific number, the error rate increases. Therefore, the choice of ensemble size is affected by the desired accuracy.

- **Computational cost**. Increasing the number of classifiers usually comes with the increase in computational cost and incomprehensibility. Hence, there often is a limit size for an ensemble that is set by users.

- **The nature of the classification problem**. In some methods, the characteristics of the classification problem have an effect on ensemble size. For example, in the *Error-Correcting Output Coding* algorithm (ECOC) suggested by Dietterich and Bakiri [21], the ensemble size is determined by the number of classes.

- **Number of processor available**. In some independent methods, such as Random Forest and Bagging or Wagging, classifiers can be trained parallel. Thus, the number of processors available can be put as an upper bound on the number of classifiers.

As mentioned above, accuracy is usually regarded as the most crucial factor that affects the decision of ensemble size. Many methods have been presented with the purpose of determining the ensemble size so that the performance of the ensemble is maximized. Rokach and Maimon [45] classified those methods into three types as follows.

3.3.1 Pre-Selection of the Ensemble size

In this method, the number of classifiers is predefined by users. For example, Random Forest algorithm allows users to set the number of decision trees used to build the forest. In other cases, such as ECOC algorithm, the ensemble size is set based on the nature of the classification problem. Pre-selection is known as the simplest method to determine the ensemble size.

3.3.2 Selection of the Ensemble size while training

The central idea behind the *selection of the ensemble size while training* method is that whenever there is a new classifier, the algorithm needs to consider the contribution of the new classifier to the ensemble. If the performance of the ensemble does not increase significantly, the process of extending the ensemble stops, and the ensemble is returned.

Banfield et al. [3] proposed an algorithm to decide when a sufficient number of classification trees have been created for an ensemble. First, the out-of-bag error graph is smoothed with a sliding window in order to reduce the variance. They choose a window size of 5. The algorithm then takes windows of size 20 on the smoothed data points and determines the maximum accuracy within that window. The process is repeated until the maximum accuracy within the current window no longer increases. At this point, the algorithm stops and returns the ensemble with the maximum raw accuracy from within that window. Figure 3.3 [3] describes the algorithm in detail.

3.3.3 Post Selection of the Ensemble size

Sharing the same perspective with pruning techniques in decision tree, *post selection of the ensemble size* methods allow the ensemble grow freely and then prune the ensemble to reduce its size and make it more effective. Margineantu and Dietterich [29] experimentally indicated that pruned ensembles may obtain a similar accuracy performance as the original ensemble. Followings are two types of post selection method: *pre-combing pruning* methods and *post-combining pruning* methods.

Algorithm 1 Algorithm for deciding when to stop building classifiers

1: $SlideSize \Leftarrow 5$, $SlideWindowSize \Leftarrow 5$, $BuildSize \Leftarrow 20$
2: $A[n] \Leftarrow$ Raw Ensemble accuracy with n trees
3: $S[n] \Leftarrow$ Average Ensemble accuracy with n trees over the previous SlideWindowSize trees
4: $W[n] \Leftarrow$ Maximum smoothed value
5: **repeat**
6: Add ($BuildSize$) more trees to the ensemble
7: $NumTrees = NumTrees + BuildSize$
//Update $A[]$ with raw accuracy estimates obtained from out-of-bag error
8: **for** $x \Leftarrow NumTrees - BuildSize$ to $NumTrees$ **do**
9: $A[x] \Leftarrow$ VotedAccuracy($Tree_1 \ldots Tree_x$)
10: **end for**
//Update $S[]$ with averaged accuracy estimates
11: **for** $x \Leftarrow NumTrees - BuildSize$ to $NumTrees$ **do**
12: $S[x] \Leftarrow$ Average($A[x - SlideSize] \ldots A[x]$)
13: **end for**
//Update maximum smoothed accuracy within window
14: W[$NumTrees/BuildSize - 1$] $\Leftarrow$ max($S[NumTrees - BuildSize]...S[NumTrees]$)
15: **until** ($W[NumTrees/BuildSize - 1] \leq W[NumTrees/BuildSize - 2]$)
16: Stop at tree $argmax_j(A[j] \mid j \in [NumTrees - 2 * BuildSize]...[NumTrees - BuildSize])$

Figure 3.3: Sample Arbiter Tree

Pre-combining Pruning

Pre-combing pruning is a method in which classifiers are chosen to be added into the ensemble before performing the combination step. The algorithm uses greedy forward-search methods to choose classifiers. According to Prodromidis et al. [39] , instead of relying just on one criterion to choose the "best" base classifiers, the pruning algorithms can employ several metrics. They suggested two methods for pre-combining pruning, which are *Diversity-based pruning* algorithm and *Coverage/Specialty-based pruning* algorithm.

In the Diversity-Based pruning algorithm, the diversity matrix d is computed, where each cell d_{ij} contains the ratio of the instances of the validation set for which classifiers C_i and C_j give different predictions. The algorithm works iteratively and in each loop, the classifier that is most diverse to the classifiers chosen so far is added into the selected set, starting with the most accurate base classifier. The loop is terminated when the N most diverse classifiers are chosen, where N is a parameter depending on some factors such as minimum system throughput, memory constraints or diversity thresholds.

The Coverage/Specialty-Based algorithm also works iteratively. It combines the coverage metric and one of the instances of the specialty metric. First, the algorithm chooses the most accurate classifier respect to the specialty metric for a particular target class on the validation set. After that, in each loop, classifiers with the best performance on the examples that the previously chosen classifiers failed to cover are selected and added into the selected set. The iteration ends when there is no more example to cover. The algorithm repeats the selection process for a different target class.

Post-combining Pruning

Assuming that classifiers are combined using a meta-combination method, contrary to the Pre-combining pruning, Post-combing pruning is considered as a backwards selection

method because it prunes a meta-classifier after it is constructed basing on their contribution to the collective. The algorithm starts with all available classifiers or with the classifiers selected by pre-combining pruning, the it iteratively tries to remove classifiers without degrading predictive performance. Following we list some among many proposed methods in this pruning family.

The *Cost complexity post-training pruning* algorithm was presented by Prodromidis et al. [39]. This algorithm consists of three phases. n the initialization phase, by applying a decision tree algorithm to the data set composed by the the meta-classifier's training set and the meta-classifier's predictions on the same set, the algorithm computes a decision tree model of the meta-classifier. The decision tree reveals the irrelevant classifiers that do not participate in the splitting criteria. Those classifiers are then pruned. In the next phase, the number of selected base classifiers continues to be reduced, according to the restrictions imposed by the available system resources or the runtime constraints. The algorithm utilizes minimal cost complexity pruning method to prune the decision tree, which leads to the reduction in size of the ensemble. Finally, in the last phase, the remaining base classifiers construct a new final ensemble meta-classifier.

The *GASEN algorithm* was developed by Zhou et al.[51] in order to build selective ensemble for neural networks. The purpose of this algorithm is to show that the appropriate neural networks for composing an ensemble can be effectively selected from a set of available neural networks. GASEN first trains a number of neural networks and then assigns a random weight to each of the networks. Next, it uses a genetic algorithm to evolve those weights so that they can characterize to some extent the fitness of the component learners in joining the ensemble. Finally, classifiers whose weights are higher than a predefined threshold are chosen to constitute the ensemble. Zhou et al. empirically showed that GASEN can generate neural network ensembles with smaller sizes but stronger generalization ability comparing to some popular ensemble methods such as Bagging and Boosting.

GASEN-b, which is a revised version of the GASEN algorithm, was proposed by Zhou and Tang [50]. GASEN-b is an extended version focusing on cases in which decision trees are used as component learners. Their experiments showed that an ensemble built by GASEN-b algorithm, which selects some of trained C4.5 decision trees to constitute an ensemble, may be not only smaller in the size but also stronger in the generalization than ensembles generated by non-selective algorithms [50]. Compared to the earlier version, GASEN-b is modified in the manner of classifiers representation. Instead of assigning a weight to each component learners, then selecting the learners according to the evolved weights, the new algorithm employs a bit string to indicate whether each classifiers is presented in the final ensemble. Then, the bit string is evolved to select final component learners. The use of bit representation can get rid of the need of manually setting the threshold for evolved weights. Moreover, because evolving shorter strings is much faster than longer ones, GASEN-b may be faster compared to the GASEN algorithm.

Prodromidis et al. [39] also suggest a method called *Correlation metric and pruning*. In this method, at first, the base classifier with the least correlation to the initial meta-classifier is removed. The algorithm then builds a new meta-classifier with the remaining classifiers, and continues to identify and removes the next least correlated base classifier. The process is repeated until enough base classifiers are pruned.

Chapter 4

Random Forest

Random forest is one of the most well-known ensemble algorithms that uses decision tree as base classifier. The construction a random forest conforms to the general process of building an ensemble, which consists of three main following phases.

1. **Gaining ensemble diversity** - Random forest algorithm gains ensemble diversity by manipulating training sets. A list of learning sets is created using the bootstrap sampling method.

2. **Constructing base classifiers** - Random forest employs the same inducer, which is random tree, on different training sets generated in the previous step to build base classifiers. In detail, at each node, a small group of input attributes is selected randomly. The size of the group can be predefined by users, but usually it is chosen as the greatest integer that is not greater than $log_2 M + 1$, where M is the number of input attributes. Next, the best attribute or the best split point would be selected to split on. All those trees are not pruned.

3. **Combining base classifiers** - The Majority voting method is utilized in the Random forest algorithm.

Breiman (2001) [12], the "father" of Random Forest, defined it as follows.

> A random forest is a classifier consisting of a collection of tree-structured classifiers $\{h(x, \Theta_k), k = 1, ...\}$ where the $\{\Theta_k\}$ are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input x.

In other words, building a random forest comprises the task of generating random vectors to grow an ensemble of trees and letting those trees vote for the most popular class.

The error rate of a forest depends on the strength of individual decision tree classifiers and the correlation among trees [12]. Increasing the strength of the individual trees increases the accuracy of the forest while increasing the correlation increases the error rate. We can notice that if one or a few input variables are very strong predictors, theses features will be selected in many trees, which causes them to become correlated. Therefore, to avoid correlation among trees, random forest uses a modified tree learning algorithm which randomly selects a subset of features instead of all features to find a best split

at each node. This is the only thing that makes random forest differ from the Bagging algorithm for trees.

The following are some noteworthy concepts related to a random forest.

4.1 Noteworthy concepts

4.1.1 The out-of-bag (oob) error estimate

In the forest building process, when bootstrap sample set is drawn by sampling with replacement for each tree, about one-third of the cases are left out and not used in the construction of that tree. This set of cases is called Out-of-bag data. Each tree has its own OOB data set which is used for calculating the error rate for an individual tree. To get the oob error rate of a whole random forest, put each case left out in the construction of the kth tree down the kth tree to get a classification. Take j to be the class that gets most of the votes every time case n is oob. The proportion of times that j is not equal to the true class of n averaged over all cases is the oob error estimate.

4.1.2 Variable importance

Random forests can be used to rank the importance of variables (features) in a regression or classification problem. The following steps were described in [12].

- In every tree grown in the forest, put down the oob cases and count the number of votes cast for the correct class
- To measure the importance of variable m, randomly permute the values of variable m in the oob cases and put theses cases down the tree.
- Subtract the number of votes for the correct class in the perturbed data from the number of votes for the correct class in the original data. The average of this number over all trees in the forest is the raw importance score of variable m.

Variable which produce large values for this score are ranked as more important than variables which produce small values.

4.1.3 Proximity matrix

Let N be the number of cases in the training set. A proximity matrix is an NxN matrix, which gives an intrinsic measure of similarities between cases. At each tree, put all cases (both training and oob) down the tree. If case i and case j both land in the same terminal node, increase the proximity between i and j by one. At the end of the run, the proximities are divided by the number of trees in the run. The proximity between a case and itself is set equal to one.

Each cell in the proximity matrix shows the proportion of trees over which each pair of observations falls in the same terminal node. The higher the proportion is, the more alike those observations are, and the more proximate they have.

Proximity matrix can be used to replace missing values for training and test set. It can also be employed to detect outliers. The following sections will illustrate how missing values are replaced and outliers are detected using the proximity matrix.

Missing value replacement

There are two ways which can be used to replace missing values in random forest. The first way is fast, simple and easy to implement. To be specific, if the m'th variable of case n is missing and it is numeric, it is replaced with the median of all values of this variable in the same class, say j, with case n. On the other hand, if the mth variable is categorical, it is replaced with the most frequent non-missing value in class j.

A more advanced algorithm capitalizes on the proximity matrix. This algorithm is computationally more expensive but more powerful. It starts by imputing missing values using the first algorithm, then it builds a random forest with the completed data. The proximity matrix from the random forests is used to update the imputations of the missing values. For numerical variable, the imputed value is the weighted average of the non-missing cases, where the weights are the proximities. For categorical variable, the imputed value is the category with the largest proximity. So, by following this algorithm, cases more similar to the case with the missing data are given greater weight.

Outliers

Outliers are cases that are removed from the main body of the data [12]. The proximity matrix can be used to detect outliers. In other words, an outlier in class j is a case whose proximities to all other cases in class j are generally small.

4.2 Related work

There have been many research works in the area of random forest aiming at improving accuracy, or performance, or both. According to the idea that to have a good ensemble, base classifiers need to be diverse and accurate, whereas random selection of attributes makes individual trees weak, Praveen Boinee et al. (2008) [9] proposed Meta Random Forest. The central idea behind the algorithm is to use random forest themselves as base classifiers for making ensembles [9]. Meta random forests are generated by both bagging and boosting approaches. The performances of those two new models were tested and compared with the original random forest algorithm. Among the three approaches, bagged random forest gives the best results.

In the original random forest, after a subset of attributes is randomly selected, one of the attributes in the subset is chosen to be a split point based on its Gini index score. However, according to Robnik and Sikonja [44], Gini index can not detect strong conditional dependencies among attributes. The reason for the deficiency of Gini index is that it measures the impurity of the class value distribution before and after the split on the evaluated attribute. In this way it assumes the conditional (upon the class) independence of attributes, evaluates each attribute separately and does not take the context of other attributes into account. Their solution was to use five different attribute measures: Gini index, Gain ratio, ReliefF, MDL and DKM. One-fifth of trees in the forest were built using one of those attribute measures. Among those measures, ReliefF is not impurity based. With this new method, they obtained better results on some data sets. However, the differences were not significant.

Robnik and Sikonja [44] also proposed another improvement for random forest, which focuses on the manner that base classifiers are combined. They noticed that not all trees are equally responsible for incorrect classification of individual instances. This simple

observation led to the idea that it would be useful to use only some selected trees in classification. Therefore, instead of counting the number of votes and using the class with major votes to be the output class, Robnik and Sikonja suggested using weighted voting. For each instance that needs to be classified, they first find some of its most similar instances. The similarity is measured by proximity. They select t most similar training instances and classify them with each tree where they are in the out-of-bag set. For each tree in the forest they measure margin using the formula defined in [12] on these similar out-of-bag instances, that is:

$$mg(X,Y) = av_k I(h_k(X) = Y) - max_{j \neq Y} av_k I(h_k(X) = j) \tag{4.1}$$

where I(.) is the indicator function.

The trees with negative average margin are left out of classification. For final classification they use weighted voting of the remaining trees where weights are average margins on the similar instances when they are in the out-of-bag set.

Table 4.1 shows the performance of the original random forest on 17 data sets in comparison to the performance of the random forest with weighted voting.

Data set	**Original Random Forest**	**Weighted voting Random Forest**
breast-cancer	0.966	0.967
bupa	0.734	0.739
diabetes	0.762	0.770
ecoli	0.869	0.869
german-numeric	0.750	0.760
glass	0.763	0.795
ionosphere	0.937	0.940
letter	0.957	0.958
parity2	0.820	0.875
parity3	0.075	0.625
sat	0.910	0.910
segmentation	0.982	0.982
sonar	0.817	0.865
vehicle	0.750	0.755
vote	0.957	0.957
vowel	0.979	0.979
zip	0.934	0.934

Table 4.1: Random forests performance for the original algorithm and weighted voting algorithm

In a "classical" random forest induction process, decision trees are independently added into the forest, which can not guarantee that all those trees will work well together. There may be some trees which make the performance of the ensemble decrease. If those "bad" trees exist, removing them from the forest may lead to a better performance. Based on that idea, Simon Bernard et al. (2009) [6] carried out experiments to find out if it is possible to enhance the accuracy of a random forest by focusing on some particular subsets of trees.

Their work was to generate many subsets of trees with all possible sizes and then compare the performances of those subsets to the original one. Two simple classifier selection techniques were used to build subsets, they are SFS (Sequential Forward Selection) and SBS (Sequential Backward Selection). Those techniques are sub-optimal because the sequential process makes each iteration depend on the previous one, and finally not all the possible solutions are explored. However, because the main goal of their paper was not to find the optimal subset of individual classifiers among a large ensemble of trees, but rather to study the extent to which it is possible to improve the performance of a RF using a subset of trees, the optimality of the selection methods is not a priority. Those two techniques were still a good choice because of their simplicity and fast performance. At each iteration of the SFS process, each remaining classifier is added to the current subset and the one that optimizes the performance of the ensemble is retained. In the same manner, in the SBS process, each classifier of the current subset is removed, and the one for which the remaining ensemble exhibits the best accuracy is definitely discarded.

Table 4.2 shows 10 datasets which were used in Bernard et al.'s experiments. They split each dataset randomly into two parts. The two thirds of the samples was used for training and the last third for testing. Denote $T = (T_T, T_S)$ where T_T and T_S respectively stood for the training and testing set.

First, a random forest was grown from T_T, with a number L of trees fixed to 300. The value of the hyperparameter K, which denotes the number of features randomly selected at each node of the trees, was fixed to $\sqrt{M}$, where M is the dimension of the feature space.

Then, each method (SFS and SBS) was applied to generate L subsets consisting of 1 to L trees. Besides SFS and SBS, Bernard also used one more random selection method, i.e. SRS (for Sequential Random Selection). The method randomly selects trees from the original set and add them to the final subset. So, in total, there were L $\times$ 3 error rates.

Dataset	**Size**	**Features**	**Classes**
Gamma	19020	10	2
Letter	20000	16	26
Pendigits	10992	16	10
Segment	2320	19	7
Spambase	4610	57	2
Vehicle	946	18	4
Waveform	5000	40	3
Ringnorm	7400	20	2
Twonnorm	7400	20	2
Mnist	60000	84	10

Table 4.2: Datasets description

Figure 4.1 shows the results of L $\times$ 3 subsets generated by SFS, SBS and SRS methods. As we can see, although SBS and SFS are two sub-optimal methods, in each dataset, there always is a subset which outperforms the full set. Thus, the selection of subsets of trees is the promising field that can bring much better results if we can find the optimal subset of trees. Another observation is that every best subset in all datasets consists of less than

100 trees. This corresponds to less than $\frac{1}{3}$ of the total number of trees in the initial forest. Those results highlight that when a random forest is grown with a "classical" random forest algorithm such as Forest-RI, all the trees do not improve the performance, and some of them even make the ensemble do more prediction mistakes.

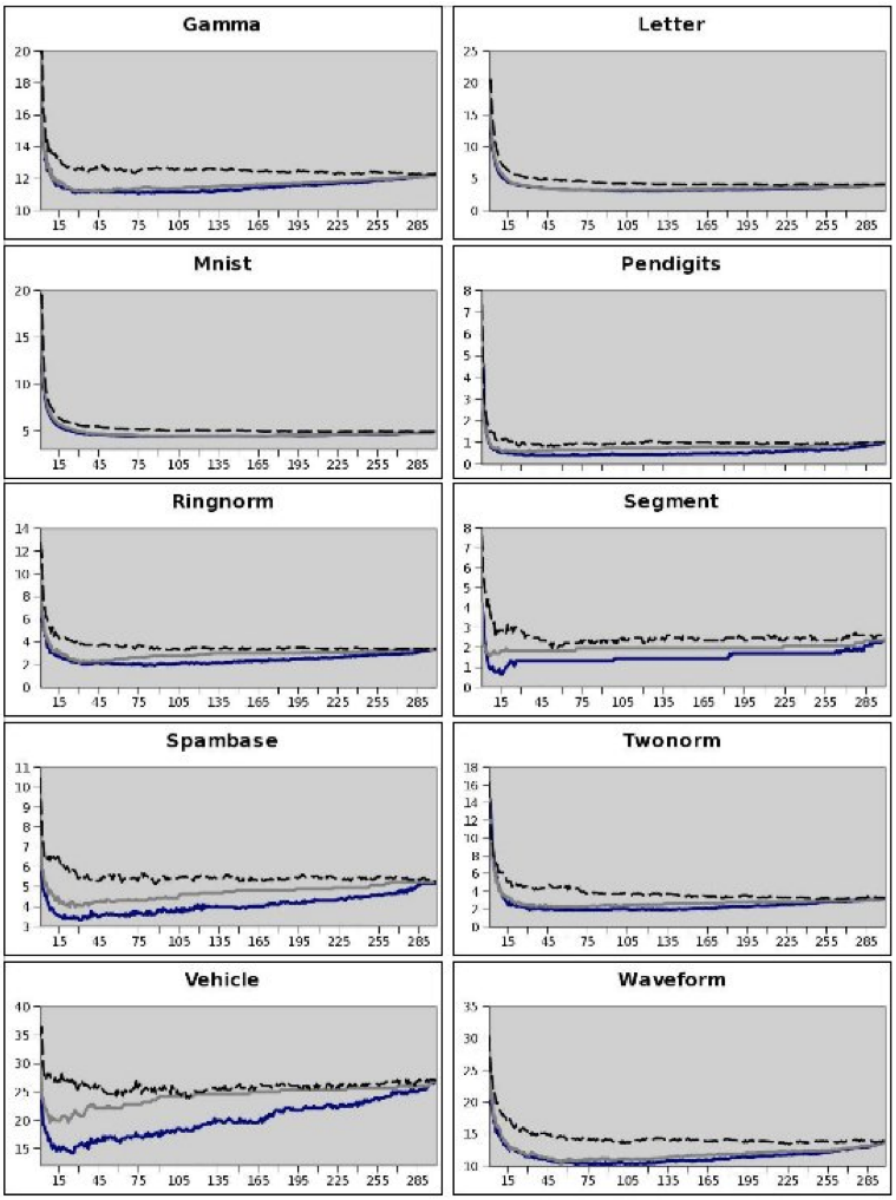

Figure 4.1: Error Rates obtained during the tree selection processes on 10 datasets, according to the number of trees in the subsets. The black curves represent the error rates obtained with SFS, the gray curves the error rates with SBS, and the dashed-line curves the error rates with SRS.

Following the previous research, Bernard et al. wanted to to identify some particular properties that are shared by these sub-forests and which properties have an effect on the error rate [7].

The *Strength* and *Correlation* are two features introduced by Breiman [12] as the key features that affect the performance of a random forest. Increasing the strength of

the individual trees increases the accuracy of the forest while increasing the correlation among trees increases the error rate. However, this result based on the assumption of a large number of trees grown in the random forest. In [6], the subsets which give the lowest error rates on employed datasets contain less than $\frac{1}{3}$ of the total number of trees in the initial forest. This means the assumption of a large number of trees in the random forest are not satisfied any more. For that reason, Bernard et al. conducted an experiment to confirm Breiman's theory with different sizes of random forest.

In particular, Bernard et al. (2011) [7] studied the relation between the ratio $\frac{\overline{\rho}}{s^2}$ and the error. They generated a pool of forests and measured for all of them the strength, the correlation and the error rate. Because their goal was to generate a large pool of sub-forests in terms of error rates, they used Genetic Algorithms (GA) for decision selection in random forest.

Table 4.3 describes 20 datasets used in the experiments. Similar to the previous research, each dataset was randomly split into 2 parts: training set and testing set, containing respectively two thirds and one third of the original dataset. A random forest was grown from T_T, with the number of tree fixed to 500. All other parameters remained the same as in the previous research. A classifier selection process using a GA was then applied to this forest. The size of sub-forests generated during this process was fixed, so that all of them could be fairly compared with each other, as the number of trees could affect the calculation of strength and correlation. The selection procedure through GA was conducted for the following sizes of sub-forests: 50, 100, 150 and 200. Concerning the GA parameters, they were fixed to the following classical values: number of generations fixed to 300, population size to 50, mutation probability to 0.01, crossover probability to 0.60 and selection proportion to 0.80. So, for each size of sub-forest (50, 100, 150, 200) and for each dataset, there were 15000 sub-forests.

Figure 4.2 illustrates the results of the experiments with 50-tree size forests. In the figure, the relation between error rate and the ratio $\frac{\overline{\rho}}{s^2}$ are shown. Each point is a sub-forest represented by its error rate and its value of $\frac{\overline{\rho}}{s^2}$. Besides, a regression line was drawn on each diagram to give a better observation. The tendencies observed on the figure can be extended to all the sub-forest sizes tested in these experiments.

We can see clearly that when the values of $\frac{\overline{\rho}}{s^2}$ decrease, error rates decrease also. This observation is consistent with Breiman's theory.

Dataset	**Size**	**Features**	**Classes**
Diabetes	768	8	2
Gamma	19020	10	2
Isolet	7797	616	26
Letter	20000	16	26
Madelon	2600	500	2
Pendigits	10992	16	10
Mfeat-factor	2000	216	10
Mfeat-fourier	2000	76	10
Mfeat-karhnen	2000	76	10
Mfeat-zernike	2000	47	10
Page-blocks	5473	10	5
Segment	2320	19	7
Musk	6797	166	2
Spambase	4610	57	2
OptDigits	5620	64	10
Vehicle	946	18	4
Waveform	5000	40	3
Digits	38142	330	10
DigReject	14733	330	2
Mnist	60000	84	10

Table 4.3: Datasets description

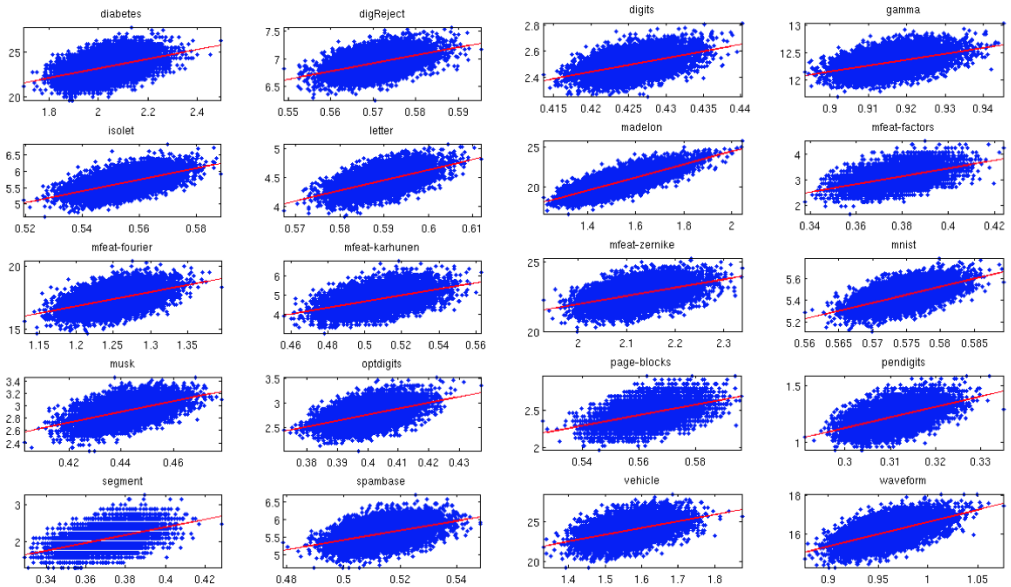

Figure 4.2: Error rates (y-axis) according to $\frac{\overline{\rho}}{s^2}$ values (x-axis) for all the sub-forests of 50 trees, obtained during the selection process. The red line is the regression line of the cloud

Chapter 5

Introduction to ADATE system

5.1 Artificial evolution

Artificial evolution is a research field inspired by Darwinian evolution. The main idea of artificial evolution is to keep a set of potential solutions of a problem and try to generate better solutions based on the old ones using some genetic operators e.g. recombination and mutation. The potential solutions are called *individuals* and a set of individuals are called *population*. Individuals in population are affected by "natural selection". Only those individuals that are good regarding to some predefined fitness measure can be kept in the population and can reproduce. The population size, in most cases, is fixed.

There are several classes of artificial evolution. The two most common ones are *Genetic algorithm* [23] and *Genetic programming*.

5.1.1 Genetic Algorithm

Genetic Algorithm is a learning method in which a solution for a problem is presented as a chromosome using a string whose elements are chosen from some alphabets. In the purpose of deciding whether a string is good or not, a fitness function is defined. It takes a string as an argument and returns a value indicating how well the string satisfied the problem criteria. Over each generation, those strings that are relatively fit compared to the other members in the population are selected to recombine. The common way to do it is just to pick a fraction f of the best strings and ignore the rest. That method is called *Truncation selection*. However, allowing some possibility of weak string is also good to produce some exploration. Another better method, called *Fitness proportional selection* is employed. The center idea of this method is to select strings with probability being proportional to theirs fitness. After parents strings are chosen, a new string can be generated by taking a part of the first parent and a part of the second. It is called *crossover*. There are three kinds of crossover, which are single point crossover, multi-point crossover and uniform crossover. After being generated, offspring also are mutated to maintain genetic diversity from one generation of a population.

5.1.2 Genetic Programming

The general algorithm of Genetic programming is similar to Genetic Algorithm. However, instead of being represented as a string, a solution in Genetic programming is a computer program, usually in some functional language. Like Genetic algorithm, a population of

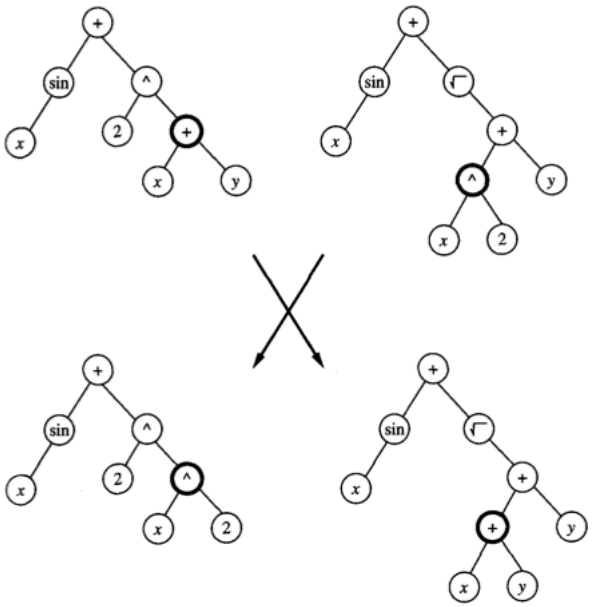

Figure 5.1: Crossover operation applied to two parent program trees (top). Crossover points (nodes shown in bold at top) are chosen at random. The subtrees rooted at these crossover points are then exchanged to create children trees (bottom)

individual (in this case, program tree) is also maintained. Over each iteration, the algorithm produces a new generation of individuals using selection, crossover and mutation. Crossover operation is performed by exchanging a subtree in the syntax tree of one parent program with the other parent program. Figure 5.1 taken from [34] illustrates a crossover operation. To calculate the fitness values, an individual, or a program, is executed on a set of training data.

5.2 Automatic programming

Computer programming is the process of of writing or editing source code. Automatic programming is a type of computer programming in which human programmers "write code that writes code". In other words, when computer programming is done by a machine, the process is called automatic programming. Human programmers only need to write a specification, then, the mechanism will generate computer code based on that specification. Automatic programming is somehow like a high-level programming language, but actually it is not. In many cases, the task of giving a solution for a problem requires much more effort than describing that problem or criticizing a solution. According to Biermann, [8], there are two reasons why researchers are interested in studying automatic programming. First, having a powerful automatic programming system which could correctly generate a program from casual and imprecise specifications for a desired target program. Second, it is widely believed that automatic programming is a necessary component of any intelligent system. Thus, it becomes a topic for fundamental research.

Rich and Waters [43] gave a "cocktail party" description of automatic programming:

> There will be no more programming. The end user, who only needs to know

> about the application domain, will write a brief requirement for what is wanted. The automatic programming system, which only needs to know about programming, will produce an efficient program satisfying the requirement.
> Automatic programming systems will have three key features: They will be end-user oriented, communicating directly with end users; they will be general purpose, working as well in one domain as in another; and they will be fully automatic, requiring no human assistance.

There will be a new era of computer science if automatic programming can achieve perfection. However, today, almost all automatic programming systems need such a long time to evolve good solutions. With the rapid development of CPU processors and parallel processing, automatic programming, hopefully, will be a promising area in the near future.

5.3 Functional programming and ML

5.3.1 Functional programming

Functional programming is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions [1]. One of the key motivations for the development of functional programming is to generate correct programs. Functional languages forbid themselves facilities which most programmers in imperative programming languages regard as standard. There is no global variables of a traditional language or the instances of objects in an object oriented language. When a value is assigned, it will not change during the execution of the program. Reassignment is not allowed. Therefore, the output of a function only depends on arguments passed to that function. If a function is called with the same arguments for many times, it always produces the same output, thus eliminating side effects. On the other hand, in imperative programming, the same expression can result in different values at different times depending on the state of the executing program. The side effects may occur and change the values of program state. This is one of the most significant differences between the functional programming and the imperative programming. The table 5.1 below summarizes some other differences between those two programming paradigms [18].

5.3.2 ML

ML is a general-purpose functional programming language developed by Robin Milner and others in the early 1970s at the University of Edinburgh, whose syntax is inspired by ISWIM. The following are some important features of ML.

Higher-Order functions

Higher-order functions are functions that take functions as arguments. Higher-order functions are supported in ML with great generality. Other languages such as C or Pascal support functions as arguments only in limited ways.

	Imperative Programming & Traditional Software Engineering	**Functional Programming & Formal Methods**
The Development Cycle	Using informal language a specification may be open to interpretation. Using appropriate testing strategies we can improve confidence - but not in any measurable way. Mistakes/bugs are common and difficult to spot and correct.	Using logic we can state the specification exactly. Using mathematics we may be able to prove useful properties of our programs. Mistakes/bugs are not common and not difficult to spot and correct.
The Development Language	Using structured programming or object oriented techniques we can reuse code. Using structured programming or object orientation we can partition the problem into more manageable chunks.	Using structured programming or object oriented techniques we can reuse code. We can partition the problem into easy to use chunks - plus there are often "higher-level" abstractions which can be made ML which would be difficult or impossible in a traditional language.
The Runtime System	The compiler can produce fast compact code taking a fixed amount of memory. Parallel processing is not possible (in general). Fancy GUI's may be added.	The memory requirements are large and unpredicatable. Parallel processing is possible. Fancy GUI's may be added, with difficulty.

Table 5.1: Comparison between Functional Programming and Imperative programming

Polymorphism

Polymorphism is the ability of a function to take arguments of various type. It allows ML programmers to write generic functions. For example, function *length* which returns the length of a list is defined

$$length \quad : \quad 'a \quad list \quad -> \quad int \tag{5.1}$$

The functions works no matter which type the arguments are. The type $'a$ can stand for any ML type.

Abstract data types

ML supports abstract data types [48] through:

- An elegant type system.
- The ability to construct new types.
- Constructs that restrict access to objects of a given type so all access is through a fixed set of operations defined for that type.

These abstract data types, called *structures*, offer the power of *classes* used in object-oriented programming languages like C++ or Java.

Recursion

A recursive function is a function which call itself either directly or indirectly. The recursive function can be used to replace loops in traditional languages. It is strongly encouraged to be used in ML. Recursive functions tend to be much shorter and clearer.For example, a factorial function could be defined as

$$\begin{array}{lllll} fun & factorial & 0 & = & 1 \\ | & factorial & n & = & n * n \quad factorial(n-1); \end{array} \tag{5.2}$$

Rule-based programming

In ML, actions are based on if-then-else rules. The core idea is to construct patterns for cases, and a value is compared with several patterns in turn. The first matching pattern causes an associated action to be compiled.

Strong typing

ML is a *strongly typed* language. That means types of all values can be determined at compile time. ML tries to figure out the unique type that each value may have. Programmers only are asked to declare a variable in case it is impossible for ML to deduce its type. Strong typing is valuable for debugging, because it allows many errors to be caught by the compiler rather than resulting in mysterious errors when the program is executed.

5.4 ADATE

ADATE (Automatic Design of Algorithms Through Evolutions) [36] is a system for general automatic programming, in a purely functional subset of the programming language Standard ML. It can be used to build a solution for a problem from scratch or improve an existing solution.

The principle of how ADATE works follows the basic idea of Genetic programming. ADATE builds and maintains a population of programs (or called individuals) during a run. The individuals are represented as expression trees. At the beginning, there is only a single individual. Then, the population expands as the evolution progresses. ADATE constructs new individuals by applying a number of atomic program transformations, called *compound program transformation*, on existing individuals in the kingdom during reproduction. The compound program transformations are composed using a number of different predefined heuristics. After being generated by a compound program transformation, a new individual is considered whether it should be added into the kingdom or not. The ADATE system decides adding one new individual into the kingdom not only based on its fitness value but also its size. Utilizing *Occam's Razor principle*, ADATE prefers small size programs. A large program will be inserted into the population only when all other smaller size programs in the population are worse than it. And after each insertion

of a new individual, larger programs which have poorer performance will be removed. As listed in [36], there are five basic forms of transformations.

1. **R** (**R**eplacement) - A part of an existing individual is replaced by a new expression. Replacement and its special use, *REQ* are two transformations that can change the semantics of a program. Among 5 transformation, it is also applied most frequently. Following is an example of R. The codes are taken from a log file when running ADATE system on Wine dataset.
   Preceding individual:

```
 fun f
    ( X0real, X1real, X2real, X3real, X4real, X5real
      X6real, X7real, X8real, X9real, X10real, X11real, X12real
    ) =
case realLess( X6real, X9real ) of
  false => (
    case
      realLess(
        X3real,
        realAdd( realAdd( X12real, X0real ), X2real )
        )      of
      true => class1
    | false => class2
    | false => (raise NA_4D9)
    )
| true => (raise D_E67800)

```

   Succeeding individual:

```
fun f
    ( X0real, X1real, X2real, X3real, X4real, X5real
      X6real, X7real, X8real, X9real, X10real, X11real, X12real
    ) =
case realLess( X6real, X9real ) of
  false => (
    case
      realLess(
        X3real,
        realAdd( realAdd( X12real, X0real ), X2real )
        )      of
      true => class1
    | false => class2
    | false => (raise NA_4D9)
    )
| true =>
case realLess( X1real, X6real ) of false => class3 | true => class2

```

   It is easy to see that the exception $raise\ D\_E67800$ in the former individual is replaced by a newly synthesized expression $case\ realLess(X1real,\ X6real)\ of\ false\ =>\ class3 \mid true\ =>\ class2$ in the latter individual.

2. **REQ** (**R**eplacement preserving **Eq**uality) - This is an R transformation that does not making the individuals evaluation value worse. REQ transformations are created by

generating many R transformations and then selecting those transformation having an equal or better evaluation value.

3. **ABSTR** (**Abstr**action) - ABSTR transformations are created by factoring out a piece of code into a function definition and replacing the original code with a function call. Auxiliary functions can be invented using this transformation. Here is an example of ABST.

   Preceding individual:

```
 fun f
    ( X0real, X1real, X2real, X3real, X4real, X5real
      X6real, X7real, X8real, X9real, X10real, X11real, X12real
    ) =
case realLess( X3real, X0real ) of
  false => (
    case realLess( X6real, X9real ) of
      false => class2
    | true => class3
    )
| true =>
case realLess( X7real, X12real ) of
  false => (raise D_1A58FA)
| true => class1
```

   Succeeding individual:

```
fun f
    ( X0real, X1real, X2real, X3real, X4real, X5real
      X6real, X7real, X8real, X9real, X10real, X11real, X12real
    ) =
let
  fun g296387 V296388 =
    case realLess( X6real, V296388 ) of
      false => class2
    | true => class3
in
  case realLess( X3real, X0real ) of
    false => g296387( X9real )
  | true =>
  case realLess( X7real, X12real ) of
    false => g296387( X1real )
  | true => class1
end
```

   In the new program, function $fun\ g296387\ V296388$ is created. An expression in the preceding function $false\ =>\ (case\ realLess(X6real, X9real) of\ false\ =>\ class2 \mid true\ =>\ class3)$ is replaced by a function call $g296387(X9real)$

4. **CASE-DIST** (**Case Dist**ribution) - This transformation takes a case expression inside a function call and moves the function call into each of the case code blocks and vice versa. Preceding individual:

```
fun f
      ( X0real, X1real, X2real, X3real, X4real, X5real
      X6real, X7real, X8real, X9real, X10real, X11real, X12real
      ) =
  let
    fun g296387 V296388 =
      case realLess( X6real, V296388 ) of
        false => class2
      | true => class3
  in
    case realLess( X7real, X12real ) of
      false => (
        case realLess( X3real, X0real ) of
          false => g296387( X9real )
        | true => g296387( X1real )
        )
    | true =>
    case realLess( X3real, X0real ) of
      false => g296387( X9real )
    | true => class1
  end

```

Succeeding individual:

```
fun f
    ( X0real, X1real, X2real, X3real, X4real, X5real
      X6real, X7real, X8real, X9real, X10real, X11real, X12real
    ) =
let
  fun g296387 V296388 =
    case realLess( X6real, V296388 ) of
      false => class2
    | true => class3
in
  case realLess( X7real, X12real ) of
    false =>
      g296387(
        case realLess( X3real, X0real ) of
          false => X9real
        | true => X1real
        )
  | true =>
  case realLess( X3real, X0real ) of
    false => g296387( X9real )
  | true => class1
end

```

The expression in the preceding function

```
case realLess( X3real, X0real ) of
          false => g296387( X9real )
        | true => g296387( X1real )

```

is replaced by

```
g296387(
        case realLess( X3real, X0real ) of
          false => X9real
        | true => X1real
        )

```

All those two expression are have the same meaning, that is

```
X3real < X0real
: ... X6real < X1real : class3
: ... X6real >= X1real : class2
X3real >= X0real
: ... X6real < X9real : class3
: ... X6real >= X9real : class2

```

5. **EMB** (**Embedding**) - This transformation changes the argument type of functions or adds arguments to functions. Below is an example of embedding, in which a new argument is added to a parent program to generate a child program.

   Preceding individual:

```
fun f
    ( X0real, X1real, X2real, X3real, X4real, X5real
      X6real, X7real, X8real, X9real, X10real, X11real, X12real
    ) =
let
  fun g3FF855 V3FF856 =
    case realLess( V3FF856, X5real ) of
      false => class3
    | true => class2
in
  case realLess( X6real, X9real ) of
    false => (
      case realLess( X3real, X0real ) of
        false => class2
      | true =>
      case realLess( X7real, X12real ) of
        false => (
          case
            case g3FF855( X2real ) of
              class1 => (raise NA_41A5C2)
            | class2 => class1
            | class3 => g3FF855( X0real )            of
            class1 => class2
          | class2 => class1
          | class3 => class2
          )
      | true => class1
      )
  | true => g3FF855( X3real )
end

```

   Succeeding individual:

```
1   fun f
2       ( X0real, X1real, X2real, X3real, X4real, X5real
3         X6real, X7real, X8real, X9real, X10real, X11real, X12real
4       ) =
5   let
6     fun g45BC9B( V45BC9C, V45BC9D ) =
7       case realLess( V45BC9D, V45BC9C ) of
8         false => class3
9       | true => class2
10  in
11    case realLess( X6real, X9real ) of
12      false => (
13        case realLess( X3real, X0real ) of
14          false => class2
15        | true =>
16        case realLess( X7real, X12real ) of
17          false => (
18            case
19              case g45BC9B( X5real, X2real ) of
20                class1 => (raise NA_41A5C2)
21              | class2 => class1
22              | class3 => g45BC9B( X5real, X0real ) of
23              class1 => class2
24            | class2 => class1
25            | class3 => class2
26            )
27        | true => class1
28        )
29    | true => g45BC9B( X5real, X3real )
30  end
31
```

In the parent program, function $fun\ g3FF855\ V3FF856$ is modified adding one argument. That function becomes $fun\ g45BC9B(\ V45BC9C,\ V45BC9D)$

In a nutshell, a procedure of how ADATE works is briefly described as follows.

1. Initiate the population with a single program given as the start program. For each program in the population, ADATE assigns a number, called *cost limit* C_P.

2. Select a program P from the population with the smallest cost limit.

3. Apply C_P compound transformations to P. So, there will be C_P new programs created.

4. Check each new program with the evaluation functions to decide whether it should be discarded or added into the population.

5. Double C_P , and repeat from step 2 until the process is terminated by users.

From a user's perspective, to evolve a solution for a problem, ADATE requires a specification file. The specification file contains data type, auxiliary functions, input data, expected output data, initial program and evaluation function.

Auxiliary functions are the functions that we believe ADATE will need to generate a good solution. In some cases, being given useful auxiliary functions, ADATE can evolve smaller and less complex program than it would have done if those functions were not

available. However, the number of auxiliary functions makes the size of the search space grow fast [5]. Therefore, there should only be necessary auxiliary functions given to ADATE.

Evaluation function is a fitness function used to grade and select solutions, called *individuals* to keep in a *kingdom*. By default, it is implemented as follows.

```
1  fun output_eval_fun( I : int, _ , Y  ) =
2      if Vector.sub( All_outputs, I ) <> Y then
3          { numCorrect = 0, numWrong = 1, grade = () }
4      else
5          { numCorrect = 1, numWrong = 0, grade = () }
6
```

The above function simply compares the output of the generated program, or called individual, with the given output which is also declared in the specification file to evaluate that individual.

Finally, the specification file also contains an initial function from which evolution will start. It usually is left empty if users want ADATE to evolve from beginning.

Chapter 6

ADATE Experiments

6.1 Design of experiments

As presented in Chapter 4, the construction a random forest consists of three main phases, including *Gaining ensemble diversity*, *Constructing base classifiers*, and *Combining base classifiers*. In order to improve the Random forest algorithm, an obvious and promising approach is to employ the ADATE system on each step in the construction process to get their "evolved versions". We, in this study, attempt to improve two out of the above steps, which are *Combining base classifiers* and *Constructing base classifiers*. There are several reasons for this choice.

Firstly, using ADATE system to improve an algorithm, especially a state-of-the-art one, is usually time-consuming. Typically, evolving such programs requires hundreds of millions of program evaluations. The reason lies in the huge number of possible combinations of transformations, especially when the solutions become large. For example, a parent program has n expressions and ADATE wants to apply an R transformation on it. There also are m new expressions that can be used to replace r in n expressions existing in the parent program. Hence, we will have $\binom{n}{r} \times m^r$ possibilities. This number will grow rapidly when the program is large, or in other words, n is large. Therefore, we want to start with a small size, fast-running but potential part. As stated before, in the combination step, random forest just simply applies majority voting method which is simple and does not require much time to perform. Hence, our first experiment concentrates on evolving the combination method.

On the other hand, constructing base classifiers is the most important part in random forest algorithm. The injection of randomness into the base classifiers is said to be the "soul" of the algorithm, since it differentiates random forest from tree-based Bagging algorithm with a better performance in most cases. Some changes in the manner of building base classifiers can greatly affect the performance of an ensemble. Therefore, improving the way in which base classifiers are generated becomes our concern in the second experiment.

In the following we will describe the designs of the two experiments in detail.

Experiment 1 - The combination of classifiers experiment

We try to improve the Majority voting algorithm using *Stacking* method (for more details on Stacking method, please refer to section 3.2.2). The choice of input attributes and the

learning algorithm at the meta-level are two most essential problems in Stacking. In this first experiment, we used classifications from base classifiers, i.e. decision random tree, together with the out-of-bag (OOB) error rates of all trees as input attributes. Initially, the learning algorithm at the meta-level shares the same behavior as Majority voting method which simply assigns the class with the most frequent vote to a new instance. The ADATE system is then required to synthesize a new meta-classifier from the initialized program. The reason we add the OOB error rates into input attributes instead of using only classifications from base classifiers is that we want to provide ADATE with more information which hopefully become useful for ADATE to generate good solutions.

The data set we choose to feed ADATE in this experiment is EEG Eye State Data Set which is taken from the UCI Machine Learning Repository. The data set contains 14980 instances. According to the description, all instances in this data set are from one continuous EEG measurement with the Emotiv EEG Neuroheadset. The duration of the measurement was 117 seconds. The eye state was detected via a camera during the EEG measurement and added later manually to the file after analysing the video frames. '1' indicates the eye-closed and '0' the eye-open state. This means there is 2 classes in the data set. The data set consists of 14 EEG values, i.e. 14 attributes, and a value indicating the eye state. All attributes are numeric.

We use nine tenths of the data set, i.e. 13482 instances, to build N base classifiers (the choice of N will be explained later). The remaining 1498 instances are classified by N learners. Those predictions together with the OOB error rates of N form input attributes for the ADATE system. The target attributes of those instances are used as training outputs for ADATE. In order to have more training input for ADATE as well as avoid overfitting over a small set, we apply a method similar to 10-fold cross-validation method. The original data set is randomly partitioned into 10 equal size subsets. In each iteration, 9 subsets are used to build trees, and the remaining one is employed to generate inputs for the ADATE system. Therefore, finally, there are 14980 sample inputs generated. Two thirds of them are use for training and another one thirds are for testing.

Another attempt to avoid overfitting is to make the sample inputs for ADATE more diverse. For a given "fold", we generate 1000 trees. For a given input to ADATE, we randomly select N of these trees where N is chosen at random between 1 and 100. Thus, each input list will contain the OOB values for different trees and have a random length between 1 and 100.

Experiment 2 - The construction of classifiers experiment

The experiment aims to improve the way in which base classifiers are generated. This process contains three main parts:

- Select randomly $\lfloor log_2 M + 1 \rfloor$, where M is the number of input attributes.
- Calculate information gain at each split point. The information gain is defined as the entropy of the whole set minus the entropy when a particular attribute is chosen.
- Return the attribute with the highest information gain for the next classification step.

First, a `f` function is developed to handle all three above steps. It will then be improved by the ADATE system. However, we suggest making a small change compared to the

original algorithm to speed up the process of building a random tree. The need of speeding up the process is motivated by the fact that the biggest issue when using the ADATE system to synthesize desirable programs is that ADATE needs a lot of time to evolve solutions for a problem. In this experiment, there are two reasons why the evolution would take much longer time than the first experiment.

- First, the size of the function that will be improved by ADATE is much bigger than the previous one. As stated before, it needs to be able to handle three tasks, including choosing a subset of attributes, calculating entropy and selecting best split points.
- Instead of building all random forests in advance, in this experiment, all trees are constructed during the process of evolving.

To speed up the process, we try to fasten the construction of a random tree, especially when handling continuous attributes. One simplified way is that instead of considering absolutely all possible values of a numerical attribute as split points, we only consider randomly 50 - 100 values and choose the best one.

In this experiment, we use 19 benchmark data sets taken from UCI Directory to create sample inputs and outputs for ADATE. Figure 6.1 shows the descriptions of those data sets.

	Number of Instances	**Number of Attributes**	**Number of Classes**	**Attribute Characteristics**
Nursery	12960	8	5	Nominal
Marketing	6876	13	9	Nominal
Tic-Tac-Toe	958	9	2	Nominal
Kr-vs-k	28056	6	18	Nominal
Contraceptive	1473	9	3	Nominal
Vehicle	846	18	4	Numerical
Wine-Quality-Red	1599	11	11	Numerical
Banana	5300	2	2	Numerical
Soybean	683	35	19	Nominal
Chess	3196	36	2	Nominal
Splice	3190	60	3	Nominal
Penbased	10992	16	10	Numerical
Phoneme	5404	5	2	Numerical
Abalone	4177	8	29	Mixed
Page-blocks	5472	10	5	Numerical
wine-quality-white	4898	11	11	Numerical
CTG	2126	21	11	Numerical
Satimage	6435	36	7	Numerical
jsbachChoralsHarmony	5665	14	102	Nominal

Table 6.1: Data sets descriptions

We used the first 8 data sets in Table 6.1, which are *Nursery*, *Marketing*, *Tic-Tac-Toe*, *Kr-vs-k*, *Contraceptive*, *Vehicle* and *Wine-Quality-Red*, to make training inputs for ADATE and the remaining 11 data sets are utilized for validation. Besides, like the

previous experiment, in order to have more training input for ADATE as well as avoid overfitting, we apply 5-fold cross-validation on each data set. Therefore, we have 40 data sets for training and 55 data sets for testing.

6.2 Implementation

To take advantage of the ADATE system to improve some parts of an algorithm, firstly the algorithm has to be implemented in Standard ML language. Moreover, the function expected to be modified by ADATE has to conform to ADATE-ML rules which is a subset of Standard ML. Another requirement for using ADATE is to write a specification that the system can employ to synthesize desirable programs. In the following subsection we will present the implementation of the Random Forest algorithm as well as carefully describe our specification files.

Implementation of the Random Forest algorithm

The implementation of the Random Forest algorithm is divided into two main modules. The first one is responsible for training a random forest, and the second one is used to classify new instances. Figure 6.1 describes the general flow chart of the training algorithm. The algorithm receives a data set as input and returns a list of decision trees. A number of decision trees in a forest is decided by users and passed into the algorithm as a parameter. Implementation of process boxes in Figure 6.1 are as follows.

- *Data reader* - implemented in function `readData` and `readClass`, which can read a data set contained in CSV files. The data is then converted it into data type `data` which is defined recursively as follows.

```
datatype data =
  dataNil | dataCons of training_instance * data
```

- *Training sets Generator* - implemented in function `createListListRand`. The generator receives a data set given by the Reader and generates a set of training sets using bootstrap sampling method.

- *Trees Builder* - implemented in function `buildTree`. Given a training set, this builder is responsible for constructing a decision tree, which is represented in the following structure.

```
datatype tree =
  leaf of class_value | dn of split_point * tree_list and tree_list =
  treeListNil | treeListCons of tree * tree_list
```

To build a random tree, the builder requires some parameters, such as training set, list of nominal attributes, number of values of each nominal attributes, number of classes, to build a decision tree. There are three main modules in the builder, including:

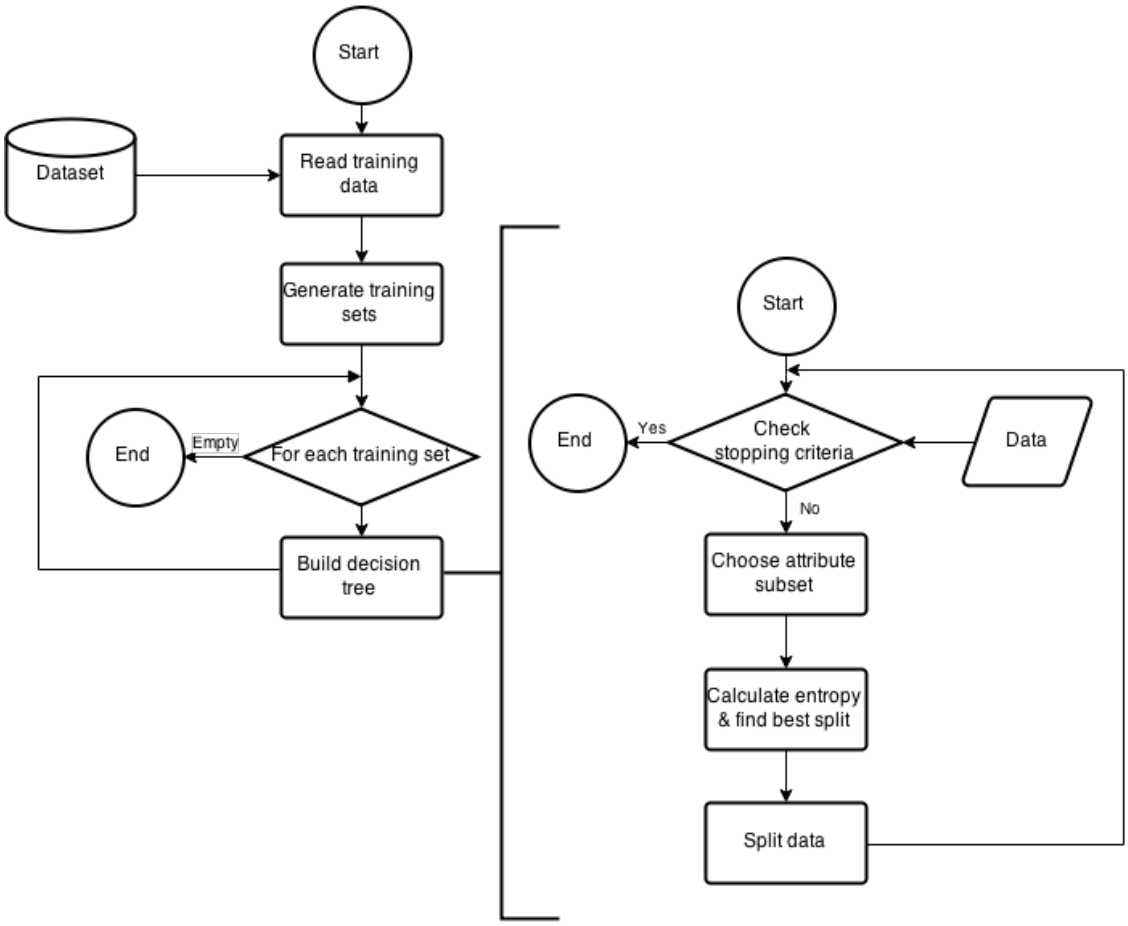

Figure 6.1: General flow chart of training algorithm for Random Forest

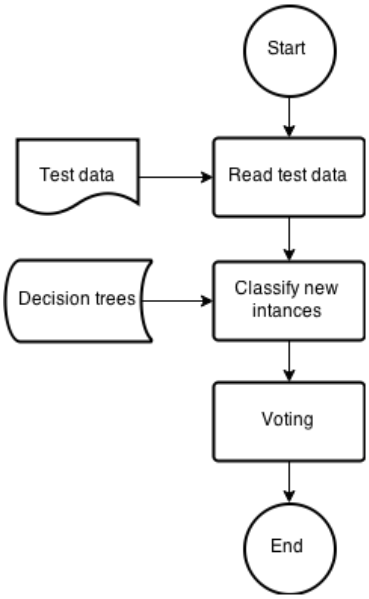

Figure 6.2: Process of classifying new instances in Random Forest

- *Checking stop criteria* - checks if the process of building a tree should stop.
- *Finding the best split point* - randomly chooses a set of attributes, calculates entropy values for each possible split point, and then returns the best split point.
- *Splitting data* - splits the data according to a given split point.

Figure 6.2 represents the process of classifying new instances in Random forest. Firstly, a *Reader* reads a test set contained in CSV files. The data is then converted it into a list of `instances` which is defined recursively as:

```
datatype attribute_value =
  nominal of int | continuous of real

datatype instance =
  instanceNil | instanceCons of attribute_value * instance
```

The new instances are classified by the list of decision trees generated in the training step. Then, those classifications are passed to the *Combiner* module, in which the classes with the most frequent votes will be assigned to the new instances.

Writing specification files

Within this section we will carefully describe our specification files used in the two experiments as well as our experiences in writing those files. For each experiment, we will focus on the `f` function, which will be modified by the ADATE system, and explain how it is started.

For more detail about the specification files, please refer to Appendix A.

Experiment 1 - The combination of classifiers experiment

The starting state of the `f` function for this experiment is relatively simple. Listing 6.1 shows the full code of the initial program . It was developed so that it will assign the class with the most frequent vote to a new instance. The `f` function receives a `tupleList` as an input and returns a `class_value`. A `tupleList` is a list of `tuple` which contains a classification of a decision tree on a given instance and its OOB error rate. Data types for `tupleList`, `tuple` and `class_value` are defined as follow.

```
datatype class_value = class of int
datatype tuple = tuple of class_value * real
datatype tupleList = tupleListNil | tupleListCons of tuple * tupleList
```

Sample Input and Output for ADATE are read from CSV files. The output files contains correct class values for all test instances. The inputs consist of a number of `tupleLists` with different sizes. In other words, each test instance is classified by different trees and different number of trees. The reason for that, as mentioned before, is to make the sample inputs diverse in order to avoid overfitting. Another notice in the input data is that OOB error rates are normalized to range $[-0.5, 0.5]$ because the `f` function can handle real numbers better than integers, especially real numbers from -0.5 to 0.5.

```
fun f( TupleList : tupleList
     ) : class_value =
  let
    fun updateVoting(
        ( Cl, LstCount ) : class_value * class_count_list
        ) : class_count_list =
      case LstCount of
        classCountListNil =>
          classCountListCons(
            classCount( Cl, 1.0 ),
            classCountListNil
          )
      | classCountListCons( CC as classCount( C, Num ), Tail ) =>
      case classEq( C, Cl ) of
        true => classCountListCons(
                  classCount( Cl, Num + 1.0 ),
                  Tail
                )
      | false => classCountListCons(
                   classCount( C, Num ),
                   updateVoting( Cl, Tail ) )
  in
  let
    fun votingHelper(
        ( TupleLst, LstCount ) : tupleList * class_count_list )
        : class_count_list =
      case TupleLst of
        tupleListNil => LstCount
      | tupleListCons( Tu as tuple( H, L), T ) =>
          votingHelper( T, updateVoting( H, LstCount ) )
  in
  let
    fun findMaxClass(
        ( LstCount, MaxCount, MaxClass ) :
        class_count_list * real * class_value
        ) : class_value =
      case LstCount of
        classCountListNil => MaxClass
      | classCountListCons( ClC as classCount( Cl, Count ), Tail ) =>
      case realLess( MaxCount, Count ) of
        true => (
      case classEq( Cl, class( ~1 ) ) of
        true => findMaxClass( Tail, MaxCount, MaxClass )
      | false => findMaxClass( Tail, Count, Cl )
        )
      | false => findMaxClass( Tail, MaxCount, MaxClass )
  in
    findMaxClass(
      votingHelper( TupleList, classCountListNil ), ~1.0, class( ~1 )
    )
  end
  end
  end
```

Listing 6.1: Initial program for Classifiers combination experiment

Experiment 2 - The construction of classifiers experiment

The `f` function in this experiment is initialized so that it is responsible for the following tasks:

- Calculate evaluation values (new entropies).
- Select a random subset of attributes.
- Select the element with min evaluation value from the remaining list and return its index.

A random subset of attributes are selected in the following manner.

Assume that there are N attributes $A1, A2, ..., AN$. Before calling `f`, that is in the code for main, we first number all attributes using N equally spaced order numbers from -0.5 to 0.5, which are already permuted. Given a threshold T, the `f` function will select a number of random attributes by taking all attributes for which the order number is less than T. The value of T is calculate by:

$$T = -0.5 + \frac{K - 0.5}{N - 1.0} \tag{6.1}$$

where K is a number of attributes out of N are to be randomly selected.

For example, assume that there are 5 attributes $[A1, A2, A3, A4, A5]$. We permute a list of 5 equally spaced order numbers from -0.5 to 0.5 and get $[0.5, -0.25, 0.0, -0.5, 0.25]$. After numbering, we get $[(A1, 0.5), (A2, -0.25), (A3, 0.0), (A4, -0.5), (A5, 0.25)]$.

If we want to choose randomly 3 attributes, by applying the above equation, we can calculate the threshold T.

$$T = -0.5 + \frac{3.0 - 0.5}{5.0 - 1.0} = 0.125 \tag{6.2}$$

That means $A2, A3, A4$ will be selected.

Before explaining in detail our initial program, we will firstly describe the data types used in the `f` function.

```
datatype rList =
  rNil | rCons of real * rList

datatype splitList =
  splitNil | splitCons of ( rList * real * real ) * splitList

datatype domainList =
  domainNil | domainCons of ( int  * splitList * real * real ) * domainList

datatype evalList =
  evalNil | evalCons of ( int  * splitList * real * real * real ) *
    evalList
```

Data type `rlist` is a list of real numbers which represents class distribution over a given data set. Given a data set and a split point, `splitList` is defined as a list of subsets which is divided from the data set according to the split point. Each subsets is represented by a combination of a `rlist`, a number of instances in the subset, and a real random number.

The random real number plays no role in the `f` function. The reason we add this filed into the data type is that we want to provide ADATE with more information which hopefully become useful for ADATE to generate good solutions. A `damainList` contains a number of possible split points. Each split point is described by an identification number, a split list created by applying the split point, a number of instances in the current data set, an order number which is generated as mentioned above. Split points from a numerical attributes share the same order number. Data type `evalList` is similar to `domainList` except that its elements has a extra field indicating the entropy value.

Listing 6.2 introduces the full code of the initial program `f` which receives a `domainList` and returns a index of the selected split point. As presented in the code, we first calculate all the entropy values for all possible split points in `evals` function. Then, a filter is applied to select a number of random attributes. Finally, `min` function will select the element with min evaluation value from the remaining list and return its index. This process seems to be inefficient because obviously we can randomly select a number of attributes before calculating entropy values. However, the target we aim to is to implement the initial function in the manner that the ADATE system can easily modify and improve. During the process of improving, ADATE may need more information than those needed in the initial program. This is the reason why we usually added more information than needed. Again, writing a specification is an art. Deciding which parts of the algorithm that can be improved or which information we should prepare in advance requires experiences as well as a bit of trial and error experimentation.

```
fun f( ( T, Ds ) : real * domainList ) : int =
let
fun evals( Ds' : domainList ) : evalList =
  let
    fun eval(
          ( I, Splits, Sum, OrderNumber ) : int * splitList * real * real
          ) : real =
    let
      fun newEntropy( ( Cards, CardSum, CardRand ) :
            rList * real * real ) : real =
        case Cards of
          rNil => 0.0
        | rCons( Card1, Cards1 ) =>
        case Card1 / CardSum of P1 =>
        newEntropy( Cards1, CardSum, CardRand )   P1 * ln P1
    in
      case Splits of
        splitNil => 0.0
      | splitCons( Split1 as ( Cards', CardSum', CardRand' ), Splits1 ) =>
          CardSum' / Sum * newEntropy Split1
    +
    eval( I, Splits1, Sum, OrderNumber )
    end
  in
    case Ds' of
      domainNil => evalNil
    | domainCons( D1' as ( I1, Splits1', Sum1, OrderNumber1 ), Ds1 ) =>
        evalCons((I1, Splits1', Sum1, OrderNumber1, eval D1' ), evals Ds1)
  end

in
let

fun filter( Es2 : evalList ) : evalList =
  case Es2 of
    evalNil => evalNil
  | evalCons( E3 as ( I3, Splits3, Sum3, OrderNumber3, Eval3 ), Es3 ) =>
  case realLess( OrderNumber3, T ) of
    true => evalCons( E3, filter Es3 )
  | false => filter Es3

in
let

fun min( Es : evalList ) : int * splitList * real * real * real =
  case Es of
    evalNil => raise NA1
  | evalCons( E4 as ( I4, Splits4, Sum4, OrderNumber4, Eval4 ), Es4 ) =>
  case Es4 of
    evalNil => E4
  | evalCons( E5 as ( I5, Splits5, Sum5, OrderNumber5, Eval5 ), Es5 ) =>
  case min Es4 of E6 as ( I6, Splits6, Sum6, OrderNumber6, Eval6 ) =>
  case realLess( Eval4, Eval6 ) of
    true => E4
  | false => E6

in

  case min( filter( evals Ds ) ) of
    E7 as ( I7, Splits7, Sum7, OrderNumber7, Eval7 ) => I7

end
end
end
```

Listing 6.2: Initial program for Classifiers construction experiment

Chapter 7

Results

7.1 Experiment 1 - Classifiers combination experiment

In this first experiment we did not succeed in finding an "improved version" which is significantly better than the original Majority Voting algorithm. However, the ADATE system simplified our initial program to the one that is less than half the size. The new program is listed in Listing 7.1.

Actually the new `f` function does the exactly the same algorithm as the original one, which finds the most common class. However, it is specified for data sets with 2 classes. That explains why it is neater than the old one. Although the new function does not make a "real improvement", it is still good news because it shows that the ADATE system can find an "optimized" and interpretable solution for a problem.

7.2 Experiment 2 - Classifiers construction experiment

During about 4 weeks running the experiment, we got 4 programs that give better performances on the 19 data sets described in Table 6.1 than the original algorithm. We have run experiments with 3 ensemble sizes (10, 20, 30 trees) and 10-fold cross validation for each data set. In the following we will explain in detail the differences between each program and the original program, as well as present the their performances tested on the 19 data sets. For full codes of those programs, please refer to Appendix B.

Improved program number 1

The first improved program makes just a small change to the original program. It changes the way the entropy of a data set is calculated. The "new entropy" (actually the "new entropy" now no longer refers to the entropy concept in Information theory) is now given by:

$$NewEntropy(D) = -\sum_{i=1}^{m} p_i ln(p_i) + T \tag{7.1}$$

where m denotes the number of classes, p_i is the probability that an instance in set D belongs to class C_i and calculated by $|C_i|/|D|$, T is the threshold that the `f` function uses to select a number of random attributes. The formula of T has been given in Equation 6.1.

```
fun f TupleList =
  let
    fun updateVoting( Cl as class( I1 ), LstCount ) =
      case LstCount of
        classCountListNil =
          classCountListCons(
            classCount( Cl, 1.0 ),
            classCountListNil
            )
      | classCountListCons(
          CC as classCount( C as class( CI ), Num ),
          Tail
          ) =
      case classEq( C, Cl ) of
        false = Tail
      | true = classCountListCons( CC, LstCount )
  in
    let
      fun votingHelper TupleLst =
        case TupleLst of
          tupleListNil = classCountListNil
        | tupleListCons( Tu as tuple( H as class( HI ), L ), T ) =
            updateVoting( H, votingHelper( T ) )
    in
      case votingHelper( TupleList ) of
        classCountListNil = (
          case TupleList of
            tupleListNil = (raise NA_C175D)
          | tupleListCons(
              VC175E as tuple( VC175F as class( VC1760 ), VC1761 ),
              VC1762
              ) =
              VC175F
          )
      | classCountListCons(
          VC1763 as classCount( VC1764 as class( VC1765 ), VC1766 ),
          VC1767
          ) =
          VC1764
    end
  end
```

Listing 7.1: New f function in Classifiers construction experiment

Thanks to the small change, the first improved version achieves slightly better performance when being tested with the 10-tree size random forest. It improves by 1.8% in total on 19 data sets. However, when the ensemble is constructed by 20 or 30 tress, the differences between the two program are not remarkable. Table 7.1 shows the performance comparison between the original program and the program number 1. Label *ori-10tr*, *ori-20tr* and *ori-30tr* stand for the performances of the original random forest algorithm built from 10, 20 and 30 trees respectively. Similarly, the performances of the first improved program are denoted as *f1,10tr*, *f1,20tr* and *f1,30tr*.

Datasets	**ori-10tr**	**f1,10tr**	**ori-20tr**	**f1-20tr**	**ori-30tr**	**f1-30tr**
Nursery	98.00	98.00	98.23	98.27	98.27	98.32
Market	29.1	28.93	29.78	29.65	30.00	30.15
Tic-Tac-Toe	91.01	91.43	94.53	94.49	96.06	96.03
Kr-vs-k	53.74	54.11	55.29	55.31	55.89	55.89
Contraceptive	47.07	46.73	48.03	47.91	48.21	48.62
Soy_bean	90.2	90.15	91.86	91.81	92.45	92.11
Chess	98.77	98.85	98.9	98.95	98.89	98.90
Splice	88.17	87.86	91.54	91.46	93.04	93.42
Vehicle	73.68	74.19	73.75	74.43	74.03	74.98
Banana	88.4	88.51	88.65	88.42	88.64	88.53
Penbased	98.57	98.54	98.88	98.89	98.99	98.96
Phoneme	89.16	89.35	90.09	89.90	90.34	90.31
Page_block	97.17	97.2	97.36	97.27	97.34	97.42
Wine-red	64.51	64.89	66.49	66.47	67.29	66.64
Wine-white	64.17	64.42	65.74	65.85	66.71	66.02
Abalone	22.42	22.87	23.18	23.06	24.01	23.78
CTG	85.32	85.37	86.06	86.28	86.83	86.70
Satimage	90.18	89.96	90.71	90.91	91.13	91.24
jsbach_chorals_harmony	71.49	71.57	71.77	71.80	71.99	71.97
SUM	1441.13	1442.93	1460.84	1461.13	1470.11	1470.00

Table 7.1: Comparison between the original program and the improved program number 1 generated by the ADATE system, tested with 10, 20 and 30-tree random forests with 10-fold cross validation

Improved program number 2

In this improved program, there are two differences from the original program. Firstly, instead of considering a number of random attributes to select the best split point, the new program investigates all available attributes. Secondly, it changes the formula of entropy. The random number *CardRand*, which is the extra information we prepared in advance, is employed in the new formula. The formula is as follows.

$$NewEntropy(D) = -\sum_{i=1}^{m} p_i ln(p_i) + CardRand \tag{7.2}$$

As we can see, although the new algorithm no longer randomly selects a subset of attributes, randomness is still injected into the algorithm, just in a different way. The

new function makes 4.8-7.8% improvement in performance in comparison with the initial program. The improvement is significant in several data sets like *Kr-vs-k* and *Splice*.

Data sets	**ori-10tr**	**f2-10tr**	**ori-20tr**	**f2-20tr**	**ori-30tr**	**f2-30tr**
Nursery	98.00	98.08	98.23	98.41	98.27	98.49
Market	29.1	28.97	29.78	29.78	30.00	30.10
Tic-Tac-Toe	91.01	91.15	94.53	94.46	96.06	96.13
Kr-vs-k	53.74	55.25	55.29	57.46	55.89	58.46
Contraceptive	47.07	46.44	48.03	47.94	48.21	47.51
Soy_bean	90.2	91.27	91.86	91.76	92.45	91.86
Chess	98.77	99.03	98.9	99.21	98.89	99.29
Splice	88.17	93.27	91.54	94.63	93.04	95.37
Vehicle	73.68	73.99	73.75	74.66	74.03	74.23
Banana	88.4	88.29	88.65	88.6	88.64	88.87
Penbased	98.57	98.87	98.88	99.12	98.99	99.14
Phoneme	89.16	89.27	90.09	90.39	90.34	90.65
Page_block	97.17	97.06	97.36	97.31	97.34	97.28
Wine-red	64.51	64.78	66.49	66.62	67.29	67.37
Wine-white	64.17	63.87	65.74	65.95	66.71	66.62
Abalone	22.42	22.79	23.18	23.53	24.01	24.11
CTG	85.32	85.13	86.06	86.33	86.83	86.42
Satimage	90.18	89.76	90.71	90.81	91.13	91.06
jsbach_chorals_harmony	71.49	71.21	71.77	71.68	71.99	71.97
SUM	1441.13	1448.48	1460.84	1468.65	1470.11	1474.94

Table 7.2: Comparison between the original program and the improved program number 2 generated by the ADATE system, tested with 10, 20 and 30-tree random forests with 10-fold cross validation

Improved program number 3

The program has shown clear improvements over the original random forests on most of the data sets (16/19 data sets). The program increases the performances of the classifiers by 7.6-14.3% in total (Table 7.3).

Similar to the previous program, in this improved program, all attributes are taken into consideration to select the best split point. The measurement used to choose the best split is also modified and given by the following formula.

$$NewEntropy(D) = -\sum_{i=1}^{m} p_i^2 + tanh(tanh(-0.472542806823)) * CardRand \tag{7.3}$$

Suppose the attribute A is now considered to be the split point and A has v distinct values $\{a_1, a_2, ..., a_v\}$. Attribute A can be used to split D into v subsets $\{D_1, D_2, ...Dv\}$ where D_i consists of instances in D that have outcome a_j. The "new entropy" is measured by the formula below.

$$Entropy_A(D) = tanh(tanh(\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Entropy(D_j) + \ 0.419265635596)) \tag{7.4}$$

Improved program number 4

The improved program number 4 is another good program we have got so far. It makes totally 8.12%-11.95% improvement in performance in comparison with the initial program. The new program is exactly similar to program number 3, except for the constant number used in Equation 7.3. The formula is as follows.

$$NewEntropy(D) = -\sum_{i=1}^{m} p_i^2 + tanh(tanh(-0.470030306823)) * CardRand \qquad (7.5)$$

Results of the program is listed in Table 7.4.

Data set	**ori-10tr**	**f3-10tr**	**ori-20tr**	**f3-20tr**	**ori-30tr**	**f3-30tr**
Nursery	98.00	98.03	98.23	98.47	98.27	98.44
Market	29.10	29.10	29.78	29.73	30.00	30.34
Tic-Tac-Toe	91.01	92.72	94.53	95.26	96.06	96.17
Kr-vs-k	53.74	56.01	55.29	59.09	55.89	60.46
Contraceptive	47.07	47.14	48.03	47.32	48.21	48.39
Soy_bean	90.20	91.32	91.86	91.47	92.45	91.96
Chess	98.77	99.16	98.90	99.24	98.89	99.34
Splice	88.17	94.33	91.54	95.15	93.04	95.31
Vehicle	73.68	74.23	73.75	74.62	74.03	74.51
Banana	88.40	88.53	88.65	88.79	88.64	88.71
Penbased	98.57	98.78	98.88	99.02	98.99	99.11
Phoneme	89.16	89.32	90.09	90.41	90.34	90.51
Page_block	97.17	97.12	97.36	97.22	97.34	97.34
Wine-red	64.51	65.37	66.49	66.05	67.29	66.93
Wine-white	64.17	64.74	65.74	66.09	66.71	66.25
Abalone	22.42	23.08	23.18	23.90	24.01	24.37
CTG	85.32	84.82	86.06	86.11	86.83	86.31
Satimage	90.18	90.13	90.71	91.00	91.13	90.99
jsbach_chorals_harmony	71.49	71.51	71.77	71.91	71.99	72.30
SUM	1441.13	1455.47	1460.84	1470.86	1470.11	1477.75

Table 7.3: Comparison between the original program and the improved program number 3 generated by the ADATE system, tested with 10, 20 and 30-tree random forests with 10-fold cross validation

Dataset	**ori-10tr**	**f4,10tr**	**ori-20tr**	**f4-20tr**	**ori-30tr**	**f4-30tr**
Nursery	98.00	98.05	98.23	98.46	98.27	98.52
Market	29.10	28.85	29.78	29.89	30.00	30.30
Tic-Tac-Toe	91.01	92.54	94.53	95.30	96.06	96.31
Kr-vs-k	53.74	56.17	55.29	59.09	55.89	60.09
Contraceptive	47.07	47.57	48.03	47.17	48.21	48.10
Soy_bean	90.20	91.27	91.86	91.23	92.45	92.21
Chess	98.77	99.25	98.90	99.33	98.89	99.32
Splice	88.17	94.16	91.54	95.17	93.04	95.36
Vehicle	73.68	73.83	73.75	75.26	74.03	74.51
Banana	88.40	88.42	88.65	88.62	88.64	88.84
Penbased	98.57	98.83	98.88	99.07	98.99	99.11
Phoneme	89.16	89.59	90.09	90.52	90.34	90.47
Page_block	97.17	97.07	97.36	97.22	97.34	97.34
Wine-red	64.51	64.45	66.49	66.26	67.29	66.97
Wine-white	64.17	63.65	65.74	65.54	66.71	66.77
Abalone	22.42	22.97	23.18	24.45	24.01	24.60
CTG	85.32	85.18	86.06	86.31	86.83	86.37
Satimage	90.18	89.74	90.71	90.65	91.13	90.85
jsbach_chorals_harmony	71.49	71.49	71.77	71.87	71.99	72.21
SUM	1441.13	1453.08	1460.84	1471.39	1470.11	1478.23

Table 7.4: Comparison between the original program and the improved program number 4 generated by the ADATE system, tested with 10, 20 and 30-tree random forests with 10-fold cross validation

Chapter 8

Conclusion and Future work

8.1 Conclusion

In this thesis we aims at two main purposes: introducing the Random Forest algorithm, and conducting ADATE experiments to improve it.

The first purpose is fulfilled in Chapter 2, 3 and 4. In those chapters, we first introduced the Decision Tree predictor. It is a predictive model from which a random forest model is constructed. Besides, the Ensemble learning, which is the general model of the Random Forest, was also investigated. Based on the understanding of Decision Tree and Ensemble learning methods, the Random Forest algorithm is presented. This background knowledge is extremely useful for us to conduct ADATE experiments, especially in the task of deciding which part of the random forest algorithm we could improve.

In this thesis, we conducted two experiments. In the first experiment, we attempted to improve the Majority voting algorithm using Stacking method. Input attributes for the ADATE system are the classifications from base classifiers and the OOB error rate of all trees. In this experiment, we did not succeed in finding an "improved version" which is significantly better than the original one. However, the ADATE system simplified our starting program to the one that is less than half size and specified for data sets with 2-value target attribute.

The second experiment concentrated on improving the way in which base classifiers, i.e. random trees, are generated. We have got 4 programs that give better performances than the original algorithm. The best program increases the performances of the classifiers on 19 data sets by 7.6-14.3% in total. All the improved versions change the way that the entropy of a data set is calculated. Most of the generated programs consider all available attributes to select the best split points, instead of only investigating a number of random attributes. However, randomness is still injected into the algorithm by using a random variable in the formula of the "new entropy".

8.2 Future work

For future experiments, we suggest these following directions.

- *Adding more extra-information in the first experiment* - In the first experiment we were not able to find a new algorithm that is remarkably better than the Major Voting. We believe the reason for our failure lies in the fact that we did not provide

the ADATE system with enough necessary information so that it can generate an effective program. The extra-information can be the sizes the base classifiers, the entropy values at each nodes, or even the whole structures of all trees. Without extra-information, it is extremely hard to generate outstanding solutions. Let take the improved algorithm proposed by Robnik and Sikonja [44] for example. In their work, for each instance that needs to be classified, they first find some of its most similar training instances and then classify them with each tree where they are in the out-of-bag set. The trees that do not show good performances are left out of classification. In the algorithm, besides the classifications from base classifiers, there are some other information required, such as the original training set, the training set of each base classifier. However, adding too much information may take the ADATE system much longer time to reach a good solution. The task of choosing which information is necessary for the system is one of our development directions in the future.

- *Improving the Gaining ensemble diversity* process - Within the scope of this thesis, we have focused on improving two out of tree main steps in the Random Forest construction process, which are Combining base classifiers and Constructing base classifiers. Improving the remaining step is also a promising direction for our future work. We can employ the ADATE system to manipulate the training set in order to generate many different training sets for classifiers, thus gaining ensemble diversity. Another possibility is to use many "improved programs" generated by ADATE in our second experiment simultaneously as base classifiers. Predictions from those classifiers then will be combined to form the final prediction.

- *Improving the Random Forest algorithm used for regression problems* - As stated before in the *Introduction*, although Random forest can be implemented for solving both classification and regression problesm, in this study, we have only concentrated on improving classification Random Forest. Improving the Random Forest algorithm used for regression problems can be one of our future work.

Bibliography

[1] *Functional Programming*, 2014.

[2] Kamal M Ali and Michael J Pazzani. Error reduction through learning multiple descriptions. *Machine Learning*, 24(3):173–202, 1996.

[3] Robert E Banfield, Lawrence O Hall, Kevin W Bowyer, and W Philip Kegelmeyer. A comparison of decision tree ensemble creation techniques. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(1):173–180, 2007.

[4] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine learning*, 36(1-2):105–139, 1999.

[5] Henrik Berg. *Evolutionary Machine Learning: Neutrality, Diversity and Application.* Unipub AS., 2009.

[6] Simon Bernard, Laurent Heutte, and Sébastien Adam. On the selection of decision trees in random forests. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 302–307. IEEE, 2009.

[7] Simon Bernard, Laurent Heutte, and Sbastien Adam. A study of strength and correlation in random forests. In De-Shuang Huang, T. Martin McGinnity, Laurent Heutte, and Xiao-Ping Zhang, editors, *ICIC (3)*, volume 93 of *Communications in Computer and Information Science*, pages 186–191. Springer, 2010.

[8] A.W Biermann. Automatic programming. In *Encyclopedia of artificial intelligence (Wiley)*, pages 18–35, 1992.

[9] Praveen Boinee, Alessandro De Angelis, and Gian Luca Foresti. Meta random forests. 2(6):1039 – 1048, 2008.

[10] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth and Brooks, Monterey, CA, 1984. new edition [**?**]?

[11] Leo Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, August 1996.

[12] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[13] Bruce G. Buchanan and Edward H. Shortliffe. *Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project (The Addison-Wesley Series in Artificial Intelligence).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[14] Philip K Chan and Salvatore J Stolfo. Experiments on multistrategy learning by meta-learning. In *Proceedings of the second international conference on information and knowledge management*, pages 314–323. ACM, 1993.

[15] Philip K Chan and Salvatore J Stolfo. Learning arbiter and combiner trees from partitioned data for scaling machine learning. In *KDD*, volume 95, pages 39–44, 1995.

[16] Phillip K Chan, Salvatore J Stolfo, et al. Toward parallel and distributed learning by meta-learning. In *AAAI workshop in Knowledge Discovery in Databases*, pages 227–240, 1993.

[17] Nitesh V Chawla, Lawrence O Hall, Kevin W Bowyer, and W Philip Kegelmeyer. Learning ensembles from bites: A scalable and accurate approach. *The Journal of Machine Learning Research*, 5:421–451, 2004.

[18] Andrew Cumming. *A Gentle Introduction to ML*, 1995.

[19] Philip Derbeko, Ran El-Yaniv, and Ron Meir. Variance optimized bagging. In *In ECML 2002*, pages 60–71. Springer-Verlag, 2002.

[20] Thomas G Dietterich. Ensemble methods in machine learning. In *Multiple classifier systems*, pages 1–15. Springer, 2000.

[21] Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *arXiv preprint cs/9501101*, 1995.

[22] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[23] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley Longman Publishing Co., 1989.

[24] Lars Kai Hansen and Peter Salamon. Neural network ensembles. *IEEE transactions on pattern analysis and machine intelligence*, 12(10):993–1001, 1990.

[25] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[26] Kenji Kira and Larry A. Rendell. A practical approach to feature selection. In *Proceedings of the Ninth International Workshop on Machine Learning*, ML92, pages 249–256, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[27] Anders Krogh, Jesper Vedelsby, et al. Neural network ensembles, cross validation, and active learning. *Advances in neural information processing systems*, pages 231–238, 1995.

[28] Yong Liu and Xin Yao. Ensemble learning via negative correlation. *Neural Networks*, 12(10):1399–1404, 1999.

[29] Dragos D Margineantu and Thomas G Dietterich. Pruning adaptive boosting. In *ICML*, volume 97, pages 211–218. Citeseer, 1997.

[30] Saeed Masoudnia and Reza Ebrahimpour. Mixture of experts: a literature survey. *Artificial Intelligence Review*, 42(2):275–293, 2014.

[31] Prem Melville and Raymond J Mooney. Constructing diverse classifier ensembles using artificial training examples. In *IJCAI*, volume 3, pages 505–510. Citeseer, 2003.

[32] ChristopherJ. Merz. Using correspondence analysis to combine classifiers. *Machine Learning*, 36(1-2):33–58, 1999.

[33] John Mingers. Expert systems-rule induction with statistical data. *Journal of the operational research society*, pages 39–47, 1987.

[34] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[35] Tim Niblett and Ivan Bratko. Learning decision rules in noisy domains. In *Proceedings of Expert Systems' 86, The 6Th Annual Technical Conference on Research and development in expert systems III*, pages 25–34. Cambridge University Press, 1987.

[36] Roland Olsson. Inductive functional programming using incremental program transformation. *Artif. Intell.*, 74(1):55–81, March 1995.

[37] David W. Opitz and Jude W. Shavlik. Generating accurate and diverse members of a neural-network ensemble. In *Advances in Neural Information Processing Systems*, pages 535–541. MIT Press, 1996.

[38] Jordan B Pollack. Backpropagation is sensitive to initial conditions. *Complex Systems*, 4:269–280, 1990.

[39] Andreas L Prodromidis, Salvatore J Stolfo, and Philip K Chan. Effective and efficient pruning of meta-classifiers in a distributed data mining system. *Knowledge Discovery and Data Mining Journal. submitted for publication*, 1999.

[40] J. Ross Quinlan. Simplifying decision trees. *International journal of man-machine studies*, 27(3):221–234, 1987.

[41] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[42] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[43] C.H. Rich and R.C. Waters. Automatic programming: Myths and prospects. *Computer*, 21:40–51, 1988.

[44] Marko Robnik-Sikonja. Improving random forests. In *Machine Learning: ECML 2004, 15th European Conference on Machine Learning, Pisa, Italy, September 20-24, 2004, Proceedings*, pages 359–370, 2004.

[45] Lior Rokach. *Data mining with decision trees: theory and applications*. World scientific, 2007.

[46] S. Shlien. Multiple binary decision tree classifiers. *Pattern Recogn.*, 23(7):757–763, July 1990.

[47] Kai Ming Ting and Ian H. Witten. Issues in stacked generalization. *Journal of Artificial Intelligence Research*, 10:271–289, 1999.

[48] Jeffrey D. Ullman. *Elements of ML programming - ML 97 edition*. Prentice Hall, 1998.

[49] David H. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[50] Zhi-Hua Zhou and Wei Tang. Selective ensemble of decision trees. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 476–483. Springer, 2003.

[51] Zhi-Hua Zhou, Jianxin Wu, and Wei Tang. Ensembling neural networks: many could be better than all. *Artificial intelligence*, 137(1):239–263, 2002.

Appendix A

Specification files

A.1 Experiment 1 - The combination of classifiers experiment

```
datatype class_value = class of int
datatype class_value_list = classValueListNil |
                            classValueListCons of class_value *
    class_value_list
datatype class_count = classCount of class_value * real
datatype class_count_list = classCountListNil |
                            classCountListCons of class_count *
    class_count_list
datatype tuple = tuple of class_value * real
datatype tupleList = tupleListNil | tupleListCons of tuple * tupleList

fun rconstLess( ( X, C ) : real * rconst ) : bool =
  case C of rconst( Compl, StepSize, Current ) => realLess( X, Current )

fun classEq( ( X as class XI, Y as class YI ) : class_value * class_value )
     : bool =
  XI = YI


fun f( TupleList : tupleList
     ) : class_value =
  let
    fun updateVoting( ( Cl, LstCount ) :
      class_value * class_count_list
    ) : class_count_list =
      case LstCount of
        classCountListNil =>
          classCountListCons( classCount( Cl, 1.0 ), classCountListNil )
      | classCountListCons( CC as classCount( C, Num ), Tail ) =>
      case classEq( C, Cl ) of
      true => classCountListCons( classCount( Cl, Num + 1.0 ), Tail )
      | false => classCountListCons( classCount( C, Num ), updateVoting( Cl
    , Tail ) )
  in
  let
    fun votingHelper( ( TupleLst, LstCount ) : tupleList * class_count_list
    ) : class_count_list =
      case TupleLst of
```

```
35          tupleListNil => LstCount
36        | tupleListCons( Tu as tuple( H, L), T ) =>
37            votingHelper( T, updateVoting( H, LstCount ) )
38    in
39    let
40      fun findMaxClass( ( LstCount, MaxCount, MaxClass ) :
41            class_count_list * real * class_value
42            ) : class_value =
43        case LstCount of
44          classCountListNil => MaxClass
45        | classCountListCons( C1C as classCount( Cl, Count ), Tail ) =>
46        case realLess( MaxCount, Count ) of
47          true => (
48        case classEq( Cl, class( ~1 ) ) of
49          true => findMaxClass( Tail, MaxCount, MaxClass )
50          | false => findMaxClass( Tail, Count, Cl )
51        )
52        | false => findMaxClass( Tail, MaxCount, MaxClass )
53    in
54      findMaxClass(votingHelper( TupleList, classCountListNil ), ~1.0, class(
       ~1 ) )
55    end
56    end
57    end
58
59
60
61  %%
62
63  type main_domain = tupleList
64  type main_range = class_value
65
66  datatype oeArg =
67      exactlyOne of ( Int.int * main_domain * main_range )
68    | allAtOnce of
69         dec *
70        ( Int.int * main_domain * main_range )List.list Option.option *
71        ( Int.int * main_domain * main_range )List.list
72
73  fun main( TupleList : tupleList ) : class_value = f( TupleList)
74
75
76  fun getFirstCharList (Cli : char list) =
77    case Cli of
78      nil => nil
79    | H::T => if H = #"," orelse H = #" " then nil else H :: getFirstCharList
       ( T )
80
81  fun parseLine (Line : char list) : real list =
82    case Line of
83      nil => nil
84    | #"," :: Tail => parseLine( Tail )
85    | #" " :: Tail => parseLine( Tail )
86    | _ =>
87      let
88        val NumStr = implode( getFirstCharList Line)
89        val Num = valOf ( Real.fromString( NumStr ) )
90      in
```

```
      Num :: parseLine( List.drop( Line, size( NumStr ) ) )
    end

exception readData_ERRORINPUT
fun readData( FileName, NLines) =
let
  fun readLines ( Fh, NLines ) =
    case ( TextIO.endOfStream Fh, NLines = 0) of
      ( _ , true ) => ( TextIO.closeIn Fh; [])
    | ( false, false ) =>
      ( parseLine( explode( valOf ( TextIO.inputLine Fh ) ) ) )
        :: readLines ( Fh, NLines   1)
    | ( true, false ) => raise readData_ERRORINPUT
in
  readLines(TextIO.openIn FileName, NLines)
end

exception readClass_ERRORINPUT
fun readClass(FileName, NLines) =
let
  fun readLines (Fh, NLines) =
    case (TextIO.endOfStream Fh, NLines = 0) of
      ( _ , true ) => ( TextIO.closeIn Fh; [] )
    | ( false, false ) =>
      valOf( Real.fromString( implode(
        getFirstCharList( explode( valOf( TextIO.inputLine Fh ) ) ) ) ) )
        :: readLines ( Fh, NLines   1)
    | ( true,false ) => raise readClass_ERRORINPUT
in
  readLines(TextIO.openIn FileName, NLines)
end

fun createOutPutADATEHelper( Lst ) =
  case Lst of
    nil => nil
  | H::T => class ( round H ) :: createOutPutADATEHelper( T )

fun readOutPutADATE( Filename, Num ) = case readClass( Filename, Num ) of
  Lst =>  createOutPutADATEHelper(Lst);

fun readInputOneLine( Lst ) = case Lst of
  nil => tupleListNil
| C::O::T => tupleListCons( tuple( class( round C ), O ), readInputOneLine(
    T ) )

fun readInputAll( LstOfLst ) = case LstOfLst of
  nil => nil
| H::T => readInputOneLine( H ) :: readInputAll( T )



val Inputs = readInputAll(readData("InputTrain.csv", 7812));


val Outputs = readOutPutADATE("OutputTrain.csv", 7812);


val Test_inputs = readInputAll(readData("InputValid.csv", 3348));
```

```
148
149
150 val Validation_outputs = readOutPutADATE("OutputValid.csv", 3348);
151
152 val All_outputs =  Vector.fromList( Outputs @ Validation_outputs )
153
154 val Funs_to_use = [
155   "false", "true",
156   "realLess", "realAdd", "realSubtract", "realMultiply",
157   "realDivide", "tanh",
158   "tor", "rconstLess",
159
160   "class", "classValueListNil", "classValueListCons",
161   "classCount", "classCountListNil", "classCountListCons",
162   "tuple", "tupleListNil", "tupleListCons",
163
164   "classEq"
165   ]
166
167 val Abstract_types = [ ]
168 val Reject_funs = []
169 fun restore_transform D = D
170 fun compile_transform D = D
171 val print_synted_program  = Print.print_dec'
172
173 val AllAtOnce = false
174
175
176 exception MaxSyntComplExn
177 val MaxSyntCompl = (
178   case getCommandOption "  maxSyntacticComplexity" of
179     NONE => 500.0
180   | SOME S => case Real.fromString S of SOME N => N
181   ) handle Ex => raise MaxSyntComplExn
182
183
184
185 val OnlyCountCalls = false
186 val TimeLimit : Int.int = 65536
187 val max_time_limit = fn () => Word64.fromInt TimeLimit : Word64.word
188 val max_test_time_limit = fn () => Word64.fromInt TimeLimit : Word64.word
189 val time_limit_base = fn () => real TimeLimit
190
191 fun max_syntactic_complexity() = MaxSyntCompl
192 fun min_syntactic_complexity() = 0.0
193 val Use_test_data_for_max_syntactic_complexity = false
194
195 val main_range_eq = op=
196 val File_name_extension = ""
197 val Resolution = NONE
198 val StochasticMode = false
199
200 val Number_of_output_attributes : Int64.int = 4
201
202
203
204 structure Grade : GRADE =
205 struct
```

```
206
207 type grade = unit
208 val zero = ()
209 val op+ = fn(_,_) => ()
210 val comparisons = [ fn _ => EQUAL ]
211 val significantComparisons = [ fn _ => EQUAL ]
212 val toString = fn _ => ""
213 val fromString = fn _ => SOME()
214
215 val pack = fn _ => ""
216 val unpack = fn _ =>()
217
218 val post_process = fn _ => ()
219
220 val toRealOpt = NONE
221
222 end
223
224
225 fun output_eval_fun( exactlyOne( I : Int.int, _ , Y ) )
226   : { numCorrect : Int.int, numWrong : Int.int, grade : Grade.grade } List.
      list = [
227   if Vector.sub( All_outputs, I ) <> Y then
228     { numCorrect = 0, numWrong = 1, grade = () }
229   else
230     { numCorrect = 1, numWrong = 0, grade = () }
231   ]
```

Listing A.1: Specification file for the Combination of Classifiers experiment

A.2 Experiment 2 - The construction of classifiers experiment

```
1
2 datatype rList =
3   rNil | rCons of real * rList
4
5 datatype splitList =
6   splitNil | splitCons of ( rList * real * real ) * splitList
7
8 datatype domainList =
9   domainNil | domainCons of ( int  * splitList * real * real ) * domainList
10
11 datatype evalList =
12   evalNil | evalCons of ( int  * splitList * real * real * real ) *
      evalList
13
14 fun rconstLess( ( X, C ) : real * rconst ) : bool =
15   case C of rconst( Compl, StepSize, Current ) => realLess( X, Current )
16
17 fun f( ( T, Ds ) : real * domainList ) : int =
18 let
19 fun evals( Ds' : domainList ) : evalList =
20   let
21     fun eval(
22           ( I, Splits, Sum, OrderNumber ) : int * splitList * real * real
23           ) : real =
24     let
```

```
25      fun newEntropy( ( Cards, CardSum, CardRand ) :  rList * real * real )
        : real =
26        case Cards of
27          rNil => 0.0
28        | rCons( Card1, Cards1 ) =>
29        case Card1 / CardSum of P1 =>
30        newEntropy( Cards1, CardSum, CardRand )   P1 * ln P1
31    in
32      case Splits of
33        splitNil => 0.0
34      | splitCons( Split1 as ( Cards', CardSum', CardRand' ), Splits1 ) =>
35          CardSum' / Sum * newEntropy Split1
36    +
37    eval( I, Splits1, Sum, OrderNumber )
38    end
39  in
40    case Ds' of
41      domainNil => evalNil
42    | domainCons( D1' as ( I1, Splits1', Sum1, OrderNumber1 ), Ds1 ) =>
43        evalCons( ( I1, Splits1', Sum1, OrderNumber1, eval D1' ), evals Ds1
        )
44  end
45
46 in
47 let
48
49 fun filter( Es2 : evalList ) : evalList =
50   case Es2 of
51     evalNil => evalNil
52   | evalCons( E3 as ( I3, Splits3, Sum3, OrderNumber3, Eval3 ), Es3 ) =>
53   case realLess( OrderNumber3, T ) of
54     true => evalCons( E3, filter Es3 )
55   | false => filter Es3
56
57 in
58 let
59
60 fun min( Es : evalList ) : int * splitList * real * real * real =
61   case Es of
62     evalNil => raise NA1
63   | evalCons( E4 as ( I4, Splits4, Sum4, OrderNumber4, Eval4 ), Es4 ) =>
64   case Es4 of
65     evalNil => E4
66   | evalCons( E5 as ( I5, Splits5, Sum5, OrderNumber5, Eval5 ), Es5 ) =>
67   case min Es4 of E6 as ( I6, Splits6, Sum6, OrderNumber6, Eval6 ) =>
68   case realLess( Eval4, Eval6 ) of
69     true => E4
70   | false => E6
71
72 in
73
74   case min( filter( evals Ds ) ) of
75     E7 as ( I7, Splits7, Sum7, OrderNumber7, Eval7 ) => I7
76
77 end
78 end
79 end
80
```

```
81
82
83 %%
84
85 type int = Int.int
86 type word = Word.word
87
88
89 datatype attribute_value =
90   nominal of int | continuous of real
91
92 datatype instance =
93   instanceNil | instanceCons of attribute_value * instance
94
95 datatype class_value =
96   class of int
97
98 datatype training_instance =
99   trainingInstance of (instance * class_value)
100
101 datatype data =
102   dataNil | dataCons of training_instance * data
103
104 datatype data_split =
105   dataSplitNil | dataSplitCons of data * data_split
106
107 datatype split_point =
108   nominalSplit of int * int | continuousSplit of int * real
109
110 datatype split_point_list =
111   splitPointListNil | splitPointListCons of split_point * split_point_list
112
113 datatype tree =
114   leaf of class_value | dn of split_point * tree_list and tree_list =
115   treeListNil | treeListCons of tree * tree_list
116
117 fun attribute_value_hash X =
118   case X of
119     nominal I => Word64.fromInt I
120   | continuous Y => realHash Y
121
122 fun instance_hash X =
123   case X of
124     instanceNil => 0wx8273639293635
125   | instanceCons( X1, Xs1 ) =>
126      list_hash( fn X => X, [ attribute_value_hash X1, instance_hash Xs1 ] )
127
128 fun class_value_hash( class I ) =  Word64.fromInt I
129
130 fun training_instance_hash( trainingInstance( I, V  ) ) =
131   list_hash( fn X => X, [ instance_hash I, class_value_hash V ] )
132
133 fun data_hash X =
134   case X of
135     dataNil => 0wx8374564297
136   | dataCons( X1, Xs1 ) =>
137      list_hash( fn X => X, [ training_instance_hash X1, data_hash Xs1 ] )
138
```

```
fun data_split_hash X =
  case X of
    dataSplitNil => 0wx937618762
  | dataSplitCons( X1, Xs1 ) =>
     list_hash( fn X => X, [ data_hash X1, data_split_hash Xs1 ] )

fun split_point_hash X =
  case X of
    nominalSplit( I1, I2  ) =>
      list_hash( fn X => X, [ Word64.fromInt I1, Word64.fromInt I2 ] )
  | continuousSplit( I1, X1 ) =>
      list_hash( fn X => X, [ Word64.fromInt I1, realHash X1 ] )

fun split_point_list_hash Xs =
  case Xs of
    splitPointListNil => 0wx728629528
  | splitPointListCons( X1, Xs1 ) =>
     list_hash( fn X => X, [ split_point_hash X1, split_point_list_hash Xs1
     ] )



fun tree_hash Xs =
  case Xs of
    leaf C => class_value_hash C
  | dn( Sp, Ys ) =>
     list_hash( fn X => X, [ split_point_hash Sp, tree_list_hash Ys ] )

and tree_list_hash Xs =
  case Xs of
    treeListNil  => 0wx618382769
  | treeListCons( X1, Xs1 ) =>
     list_hash( fn X => X, [ tree_hash X1, tree_list_hash Xs1 ] )



type main_domain =
      int * real list list * real list * int * int list * int * real *
    string list
type main_range = tree list



fun main_range_hash( Xs : tree list ) : Word64.word =
  list_hash( tree_hash, Xs )



datatype oeArg =
    exactlyOne of ( Int.int * main_domain * main_range )
  | allAtOnce of
       dec *
      ( Int.int * main_domain * main_range )List.list Option.option *
      ( Int.int * main_domain * main_range )List.list

fun createOrders (
      ( AttNums, Cur, Step ) : real * real * real
      ) : real list = case realEqual( AttNums, 0.0 ) of
```

```
195   true => nil
196   | false => Cur :: createOrders ( AttNums   1.0, Cur + Step, Step )
197
198 (* END: create random nums *)
199
200 (*  START: random permuate  *)
201 fun isMemberList ( Lst : int list, E : int) = case Lst of
202   nil => false
203 | H::T => (
204     case ( H = E ) of
205       true => true
206     | false => isMemberList ( T, E )
207 )
208
209 fun generateRandomNums (
210       ( N, Max, R, L ) : int * int * Random.rand * int list
211       ) : int list =
212   case ( N = 0 ) of
213     true => L
214   | false => (
215   case ( Random.randRange ( 0, Max ) R ) of R1 => (
216   case isMemberList ( L, R1 ) of
217     true => generateRandomNums ( N, Max, R, L )
218   | false => generateRandomNums ( N   1, Max, R, R1 :: L )
219   )
220 )
221
222 fun mapIndex2Value (
223 ( A1, L) : 'a array * int list
224 ) : 'a list = case L of
225   nil => nil
226   | H :: T => Array.sub (A1, H) :: mapIndex2Value (A1, T)
227
228 fun randomPerm (
229   ( A, R) : 'a array * Random.rand
230   ) : 'a list =
231   case generateRandomNums (
232           Array.length(A),
233           Array.length(A)   1,
234           R,
235           nil
236      ) of L => mapIndex2Value(A, L)
237
238 (*  END: random permutate  *)
239
240 (* START: Quick sort *)
241
242 exception swap_OUTOFINDEX
243 fun swap (AttVals, I, J) =
244   case I > Array.length AttVals orelse J > Array.length AttVals of
245     true => raise swap_OUTOFINDEX
246   | false =>
247       let
248         val AttTemp = Array.sub(AttVals, J )
249       in
250         Array.update(AttVals, J, Array.sub(AttVals, I));
251         Array.update(AttVals, I, AttTemp)
252       end
```

```
253
254 fun partition (Arr : (real*class_value) array, Left, Right, Pivot) =
255   case Int.compare(Left, Right) of
256     GREATER => Right
257   | EQUAL=> (
258       case Array.sub(Arr, Right) of
259         (K, _) => (
260           case K > Pivot of
261             true => Right   1
262           | false => Right
263         )
264   )
265   | LESS =>
266       case Array.sub(Arr, Left) of
267         (K, _) => (
268           case K < Pivot of
269             true => partition (Arr, Left + 1, Right, Pivot)
270           | false => (
271               case Array.sub(Arr, Right) of
272                 (K, _) => (
273                   case K > Pivot of
274                     true => partition (Arr, Left, Right   1, Pivot)
275                   | false => (swap(Arr, Left, Right);
276                               partition (Arr, Left + 1, Right   1, Pivot))
277                 )
278    ))
279
280
281 fun quickSort ( Arr : (real*class_value) array, Left, Right) =
282   case Left < Right of
283     true => (
284       let
285         val Middle = Real.ceil(Real.fromInt (Left + Right) / 2.0)   1
286         val (Pivot, _) = Array.sub(Arr, Middle)
287         val PivotIndex = partition(Arr, Left, Right, Pivot)
288         val _ = quickSort(Arr, Left, PivotIndex)
289         val _ = quickSort(Arr, PivotIndex + 1, Right)
290       in
291         Arr
292       end
293     )
294   | false =>  Arr
295
296 fun quickSortList(
297   Lst : (real * class_value) list list
298   ) : (real * class_value) array list =
299   case Lst of
300     nil => nil
301   | H :: T => quickSort( ( Array.fromList H ), 0, length H   1) ::
      quickSortList( T )
302
303 (* END: Quick sort *)
304
305 (* START: create distribution from data : real array array array
306 Ex : val a = initDistribution( [2, 3, 4, 3], Array.array(4, Array.array(0,
      Array.array(0, 0.0 ) ) ), 0, 3);
307       val dist = createDist (data1, a);
308 *)
```

```
fun initDistribution (
      ( NumAttr, A , Index, NumClass ) : int list * real array array array
    * int * int
      ) : real array array array =
  let
    fun initArrs (
            ( N, L, NumClass ) : int * real array list * int
            ) : real array array =
        case N of
          0 => Array.fromList ( L )
        | K => initArrs ( K   1, Array.array (NumClass, 0.0) :: L ,
    NumClass)
  in
    case NumAttr of
      nil => A
    | H :: T => (
      Array.update ( A, Index, initArrs ( H, nil, NumClass ) ) ;
      initDistribution ( T, A, Index + 1, NumClass )
    )
  end

(* class, attr, value are counted from 0
*)
fun updateDist (
      ( Arr, Attr, Value, Class as ( class K )
      ) : ( real array array array * int * int * class_value )
      ) : unit =
    case Array.sub ( Array.sub ( Array.sub ( Arr, Attr ) , Value ) , K ) of
      X => Array.update ( Array.sub ( Array.sub ( Arr, Attr ) , Value ) , K
    , X + 1.0 )

exception continuous_updateDistIns;
fun updateDistIns (
    ( Instance, Class, Dist, Index, LstCont ) :
      instance * class_value * real array array array * int * ( real *
    class_value ) list list
    ) : real array array array * ( real * class_value ) list list =
  case Instance of
    instanceNil => ( Dist, LstCont )
  | instanceCons ( nominal Value, Ins ) => (
      updateDist ( Dist, Index, Value, Class );
      updateDistIns ( Ins, Class, Dist, Index + 1, LstCont )
    )
  | instanceCons (continuous Value, Ins) => (
  case LstCont of H :: T => (
  case updateDistIns ( Ins, Class, Dist, Index, T ) of
    ( A1, A2 ) => ( A1, ( ( Value, Class ) :: H ) :: A2 )
  )
  )

fun createDist (
      ( Data, Dist, LstCont ) : data * real array array array * ( real *
    class_value ) list list
      ) : real array array array * ( real * class_value ) list list =
  case Data of
    dataNil => ( Dist, LstCont )
  | dataCons( T as trainingInstance ( Instance, Class ), D ) => (
  case updateDistIns ( Instance, Class, Dist, 0, LstCont ) of
```

```
362     ( A1, A2 ) => createDist ( D, A1, A2 )
363   )
364
365
366 (* END: create distribution from data : data  > real array array array *)
367
368 (* START: from distribution to domainList :
369           real array array array  > domainList
370 *)
371
372 fun arrToTuple (
373       ( A, R ) : real array * Random.rand
374       ) : rList * real * real =
375   let
376     fun realArr2rList ( (A, Index, L ) : real array * int * int ) : rList =
377       case ( Index  = L ) of
378         true => rNil
379       | false => (
380       case realEqual ( Array.sub( A, Index ), 0.0 ) of
381         true => realArr2rList ( A, Index + 1, L )
382       | false => rCons ( Array.sub( A, Index ), realArr2rList ( A, Index +
    1, L ) )
383       )
384   in
385     let
386       fun sumArr (
387           ( A, Index, Len, Sum ) : real array * int * int * real
388             ) : real =
389         case ( Index = Len ) of
390         true => Sum
391         | false => realAdd ( Array.sub( A, Index ), sumArr ( A, Index + 1,
    Len, Sum ) )
392     in (
393          realArr2rList ( A, 0, Array.length ( A ) ),
394          sumArr ( A , 0, Array.length ( A ), 0.0 ),
395          Random.randReal R   0.5
396        )
397     end
398   end
399
400 fun arrs2splitList (
401       ( A, Index, Len, R, Sum ) : real array array * int * int * Random.
    rand * real
402       ) : splitList * real =
403   case ( Index = Len ) of
404     true => ( splitNil, Sum )
405   | false => (
406   case arrToTuple ( Array.sub(A, Index), R ) of
407     R0 as ( R1, R2, R3 ) => (
408   case realEqual( R2, 0.0 ) of
409     true => arrs2splitList ( A, Index + 1, Len, R , Sum + R2)
410   | false =>
411   case arrs2splitList ( A, Index + 1, Len, R , Sum + R2) of
412     R4 as (R5, R6) =>  ( splitCons ( R0 , R5 ), R6 )
413 )
414 )
415
416 (*
```

```
Params:
  A : distribution of nominal attributes
  Index: start from 0
  Len : len of A
  R: Random seed
  OList: List of order numbers used to randomly select attributes
Return:
  domainList from Data
  remaining OList used to assign order numbers to continuous attributes
*)
fun dist2domainList (
      ( A , Index, Len , R, OList
      ) : real array array array * int * int * Random.rand * real list
      ) : domainList * real list =
  case ( Index = Len ) of
    true => ( domainNil, OList )
  | false => (
  case arrs2splitList (
         Array.sub ( A, Index ),
         0,
         Array.length ( Array.sub ( A, Index ) ),
         R,
         0.0 ) of
    (R1, R2) => (
  case OList of H :: T => (
  let
    val D = dist2domainList ( A, Index + 1, Len, R, T )
  in
    ( domainCons ( ( Int64.fromInt Index, R1, R2, H ), #1 D ), #2 D )
  end
  )
  )
)

(* END: from distribution to domainList

val orList = randomPerm( Array.fromList ( createOrders (4, ~0.5, 1.0/3.0) )
    );
val d = dist2domainList ( dist, 0, Array.length(dist), Random.rand(1,1),
    orList);

*)

(* START: create distribution for continuous attribute *)

fun distForOneSortedArr (
      ( A, Res, Index, Len ) : ( real * class_value ) array * real array *
    int * int
    ) : real array =
  case ( Index = Len ) of
  true => Res
  | false => (
  case Array.sub ( A, Index ) of
    ( V, Cl as class C ) => (
  case Array.sub ( Res, C ) of
    Count => (
      Array.update ( Res, C, Count + 1.0 );
    distForOneSortedArr ( A, Res, Index + 1, Len )
```

```
472     )
473   )
474   )
475
476 (* can not be an empty array *)
477 fun firstSplitOneContAttr (
478       ( A, Index, Len ) : ( real * class_value ) array * int * int
479       ) : int =
480   case ( Index = Len   1 ) of
481     true => Index
482   | false => (
483   case realEqual( #1 ( Array.sub ( A, Index ) ), #1 ( Array.sub ( A, Index
      + 1 ) ) ) of
484     true => firstSplitOneContAttr ( A, Index + 1, Len )
485   | false => Index
486  )
487
488 fun updateCont (
489       ( E as ( V, C as class Cl ), Dist ) : ( real * class_value ) * real
     array array
490       ) : unit =
491 let
492   val N0 = Array.sub( Array.sub ( Dist, 0 ), Cl )
493   val N1 = Array.sub( Array.sub ( Dist, 1 ), Cl )
494 in
495  ( Array.update( Array.sub( Dist, 0 ), Cl, N0 + 1.0 );
496    Array.update( Array.sub( Dist, 1 ), Cl, N1   1.0 )
497  )
498 end
499
500
501 fun initArrArr (
502       ( Num1, Num2 ) : int * int
503       ) : real array array =
504 let
505   val A = Array.array ( Num1, Array.array( 0, 0.0 ) );
506   fun updateArr ( Num ) =
507     case ( Num = 0 ) of
508       true => A
509     | false => (
510         Array.update ( A, Num   1, Array.array ( Num2, 0.0 ) );
511     updateArr ( Num   1 )
512   )
513 in
514   updateArr ( Num1 )
515 end
516
517 fun copyArrArr (
518       ( A1, A2, Index ) : 'a array array * 'a array array * int
519       ) : int =
520   case ( Index = Array.length( A1 ) ) of
521    true => 0
522    | false => (
523        Array.copy{ di = 0, dst = Array.sub( A2, Index ), src = Array.sub(
     A1, Index ) };
524        copyArrArr(A1, A2, Index + 1)
525 )
526
```

```
fun initDistOneContAttrHelper (
      ( A, Index, Len, Dist, Lst, NumClass, ValueLst ) :
    ( real * class_value ) array * int * int *
       real array array * real array array list * int * real list
      ) : real array array list * real list =
  let
    val D3' = initArrArr( 2, NumClass )
  in
  case ( Index >= Len   2 ) of
    true => ( Lst, ValueLst )
  | false => (
      updateCont ( Array.sub( A, Index + 1 ), Dist );
      copyArrArr(Dist, D3', 0);
  case realEqual( #1 ( Array.sub ( A, Index + 1 ) ), #1 ( Array.sub ( A,
    Index + 2 ) ) ) of
    true => initDistOneContAttrHelper ( A, Index + 1, Len, Dist, Lst,
    NumClass, ValueLst )
  | false =>
      initDistOneContAttrHelper (
      A, Index + 1, Len,
      Dist,
      Lst @ [ D3' ], NumClass, ValueLst @ [((#1 ( Array.sub ( A, Index + 1
    ) )) + (#1 ( Array.sub ( A, Index + 2 ) )))/2.0] )
  )
  end

fun initDistOneContAttr (
      ( A, NumClass ) :
    ( real * class_value ) array * int
      ) : real array array list * real list =
let
  val Len = Array.length ( A )
  val FirstSplit = firstSplitOneContAttr ( A, 0, Len )
  val FirstSplitValue = #1( Array.sub( A, FirstSplit ) )
  val R1 = distForOneSortedArr ( A, Array.array ( NumClass, 0.0 ), 0,
    FirstSplit + 1 )
  val R2 = distForOneSortedArr ( A, Array.array ( NumClass, 0.0 ),
    FirstSplit + 1, Len )
  val R = Array.array ( 2, Array.array( 0, 0.0 ) )
  val D2' = initArrArr( 2, NumClass )
in
  Array.update( R, 0, R1 );
  Array.update( R, 1, R2 );
  copyArrArr(R, D2', 0);
  let
    val D = initDistOneContAttrHelper ( A, FirstSplit, Len, R, [ D2' ],
    NumClass, nil )
    val D1' = #1 D
    val D2' = #2 D
  in
  case ( FirstSplit = Array.length( A )   1 ) of
    true => ( D1', FirstSplitValue :: D2' )
  | false => (
      D1',
      ((#1( Array.sub( A, FirstSplit ) ) + #1( Array.sub( A, FirstSplit + 1
    ) )) / 2.0 ) ::
        D2'
    )
```

```
578   end
579 end
580
581 (* All splits use the same random point. DomainOld is added
582 to tail of the new one
583  *)
584 fun dist2domainList2 (
585       ( A , Index, Len , R, Order, DomainOld, ContLstOld, Step, ContNum, A2
586       ) : real array array array * int * int * Random.rand * real  *
    domainList * ( int * real ) list * int * int * real array
587       ) : domainList * ( int * real ) list  =
588   case ( Index >= Len ) of
589     true => ( DomainOld, ContLstOld )
590   | false => (
591   case arrs2splitList (
592           Array.sub ( A, Index ),
593           0,
594           Array.length ( Array.sub ( A, Index ) ),
595           R,
596           0.0 ) of
597     (R1, R2) =>
598   let
599     val D = dist2domainList2 ( A, Index + 1, Len, R, Order, DomainOld,
    ContLstOld, Step, ContNum, A2 );
600     val DomainLst = #1 D;
601     val ContLst = #2 D
602   in
603     ( domainCons ( ( Int64.fromInt( Index + Step ), R1, R2, Order ),
    DomainLst), ( ContNum, Array.sub ( A2, Index ) ) :: ContLst )
604   end
605 )
606
607 fun subsetArr(
608   (A1, A2, Step, Index, Len ) : 'a array * 'a array * int * int * int
609    ) : 'a array =
610    case ( Index < Len ) of
611    true => (
612      Array.update(A2, Index, Array.sub(A1, ( Index + 1) * Step   1 ) );
613      subsetArr(A1, A2, Step, Index + 1, Len)
614   )
615    | false => A2
616
617 fun addDistContToDomain (
618       ( LstCont, CurDomain, CurMapLst, NumClass, R, OList, Step, ContNumLst
    ) :
619       (real * class_value) array list * domainList * ( int * real ) list *
    int * Random.rand * real list * int * int list
620       ) : domainList * ( int * real ) list =
621   case LstCont of
622     nil => ( CurDomain, CurMapLst )
623   | H :: T => (
624   case OList of HO :: TO => (
625   case ContNumLst of HC :: TC =>
626   let
627     val A = initDistOneContAttr ( H, NumClass ) ;
628   val A1 = Array.fromList ( #1 A );
629   val A2 = Array.fromList( #2 A );
630   val Len = Array.length( A2 );
```

```
631   val nextInt = Random.randRange (51,101);
632   val RandNum = nextInt R;
633   in (
634     case ( RandNum >= Len ) of
635       true =>
636     let
637       val Re = addDistContToDomain( T, CurDomain, CurMapLst, NumClass, R,
      TO, Array.length( A1 ) + Step, TC );
638       val DomainOld = #1 Re;
639       val MappingOld = #2 Re;
640   in
641     dist2domainList2 ( A1, 0, Len, R, HO, DomainOld, MappingOld, Step, HC,
      A2 )
642   end
643     | false =>
644   let
645     val Step1 = floor ( Real.fromInt(Len) / Real.fromInt(RandNum) )
646     val A1' = subsetArr ( A1,  Array.array(RandNum 1, Array.array(0, Array.
      array(0, 0.0))), Step1, 0, RandNum 1)
647     val A2' = subsetArr ( A2,  Array.array(RandNum 1, 0.0), Step1, 0,
      RandNum 1)
648       val Re = addDistContToDomain( T, CurDomain, CurMapLst, NumClass, R,
      TO, RandNum   1 + Step, TC )
649       val DomainOld = #1 Re;
650       val MappingOld = #2 Re;
651   in
652       dist2domainList2 ( A1', 0, RandNum 1, R, HO, DomainOld, MappingOld,
      Step, HC, A2' )
653   (*    dist2domainList2 ( A1, 0, Len, R, HO, DomainOld, MappingOld, Step,
      HC, A2 )   *)
654     end
655   )
656   end
657   )
658   )
659
660 fun makeListOfDataNil( Num : int ) : data list = case Num of
661   0 => []
662 | N => dataNil :: makeListOfDataNil( Num   1)
663
664 exception outOfBound_getValueAtrAtIndex;
665 fun getValueAttrAtIndex (instanceNil, Index) = raise
      outOfBound_getValueAtrAtIndex
666 |  getValueAttrAtIndex (instanceCons (AttValue, Ins), 0) = AttValue
667 |  getValueAttrAtIndex (instanceCons (AttValue, Ins), N) =
      getValueAttrAtIndex (Ins, N   1 )
668
669
670 fun removeAttrFromInstance(
671   ( Instance, K ) : instance * int
672   ) : instance =
673   case K of
674     0 => (
675   case Instance of instanceCons (A, I) => I
676 )
677   | L => (
678   case Instance of instanceCons (A, I) =>
679     instanceCons (A, removeAttrFromInstance ( I, K   1 ) )
```

```
680 )
681
682 fun updateListSubSets (trainingInstance(Instance, Cl), H::T, Value) =
683   case Value of
684     0 => (dataCons(trainingInstance(Instance, Cl), H))::T
685   | N => H::updateListSubSets(trainingInstance(Instance, Cl), T, Value   1
      )
686
687 fun updateListSubSetsCont (
688       trainingInstance(Instance, Cl), H::T, Value : real, ValueSplit : real
689       ) =
690   case (Value < ValueSplit) of
691     true => (dataCons(trainingInstance(Instance, Cl), H))::T
692   | false => (
693   case T of
694     Data2 :: nil => H::[(dataCons(trainingInstance(Instance, Cl), Data2))]
695   )
696
697 exception instanceNil_splitData
698 exception typeMismatch_splitData
699 fun splitData (Data, Sp, Lst) =
700   case Data of
701     dataNil => (
702   case Sp of
703     nominalSplit(Index, Value) => makeListOfDataNil(List.nth(Lst, Index))
704   | continuousSplit(Index, Value) => [dataNil, dataNil]
705   )
706   | dataCons(trainingInstance(Instance, Cl), DataTail) => (
707   case Instance of
708     instanceNil => raise instanceNil_splitData
709   | instanceCons(Att, Ins) => (
710   case Sp of
711     nominalSplit (Index, Value) => (
712   case getValueAttrAtIndex(Instance, Index) of
713     nominal K =>
714       updateListSubSets(
715         trainingInstance(removeAttrFromInstance(Instance, Index), Cl),
716         splitData(DataTail, Sp, Lst),
717         K )
718   | continuous K => raise typeMismatch_splitData
719   )
720   | continuousSplit (Index, Value) => (
721   case getValueAttrAtIndex(Instance, Index) of
722     continuous K =>
723       updateListSubSetsCont(
724         trainingInstance( Instance, Cl ),
725         splitData( DataTail, Sp, Lst ),
726         K, Value )
727   | nominal K => raise typeMismatch_splitData
728   )
729   )
730   )
731
732 (* END: Split data *)
733
734
735 (* START: build tree *)
736
```

```
737 exception noEleInList_removeEleFromList
738 fun removeEleFromList(Index, Lst) = case Index of
739   0 => tl (Lst)
740 | I => hd(Lst) :: removeEleFromList (Index   1, tl(Lst))
741
742 fun updateListAttr(Sp, Lst) = case Sp of
743   nominalSplit(Index, Value) => removeEleFromList(Index, Lst)
744 | continuousSplit(Index, Value) => Lst
745
746 fun calT ( (K, N) : real * real ) : real =
747   case realLess ( K, N ) of
748     true => ~0.5 + ( K   0.5 ) / ( N   1.0 )
749   | false => 1.0
750
751 fun createNumNilList ( Num : real ) : ( real * class_value ) list list =
752   case realEqual( Num, 0.0 ) of
753   true => []
754   | false => [] :: createNumNilList ( Num   1.0 )
755
756
757
758 fun divideTypeAtt ( Data : data ) : ( int * real ) list * int list =
759   case Data of
760   dataNil => ( nil, nil )
761   | dataCons( trainingInstance( Instance, Cl ), DataTail ) =>
762   let
763     fun divideTypeAttHelper (
764           ( Ins, Index ) : instance * int
765           ) : ( int * real ) list * int list =
766       case Ins of
767         instanceNil => ( nil, nil )
768       | instanceCons ( A as nominal N, InsTail ) =>
769       let
770         val D = divideTypeAttHelper ( InsTail, Index + 1 )
771         val NList = #1 D
772         val CDist = #2 D
773       in
774         ( ( Index, 0.0 ) :: NList, CDist )
775       end
776       | instanceCons ( A as continuous N, InsTail ) =>
777       let
778         val D = divideTypeAttHelper ( InsTail, Index + 1 )
779         val NList = #1 D
780         val CDist = #2 D
781       in
782         ( NList, Index :: CDist )
783       end
784   in
785     divideTypeAttHelper ( Instance, 0 )
786   end
787
788
789 fun findBestSplit (
790       ( Data, LstAttrDesc, NumClass, NumAttr, K, R ) : data * int list *
      int * real * real * Random.rand
791     ) : split_point =
792   let
793     val LstTypes = divideTypeAtt ( Data );
```

```
794   val NList = #1 LstTypes;
795   val CDist = #2 LstTypes;
796   val Init = initDistribution( LstAttrDesc , Array.array(List.length(
        LstAttrDesc ), Array.array(0, Array.array(0, 0.0 ) ) ), 0, NumClass);
797     val NumContAttr = NumAttr    Real.fromInt( List.length( LstAttrDesc ) );
798   val InitContLst = createNumNilList ( NumContAttr );
799   val Dist = createDist ( Data, Init, InitContLst ); (* TO DO *)
800   val DistNom = #1 Dist;
801   val DistCont = #2 Dist; (* (real, class) list list *)
802     val OrList = randomPerm( Array.fromList ( createOrders ( NumAttr, ~0.5,
        1.0/(NumAttr     1.0) ) ), R );
803     val D = dist2domainList ( DistNom, 0, Array.length( DistNom ), R,
        OrList);
804   val DNom = #1 D; (* Domain from Nominal dist *)
805   val OrListCont = #2 D; (* remaining Orlist that has not been used *)
806   val All = addDistContToDomain( quickSortList( DistCont ), DNom, nil,
        NumClass, R,  OrListCont, Array.length( DistNom ), CDist )
807   val DomainAll = #1 All;
808   val MappingLst = Array.fromList( NList @ ( #2 All ) );
809   in
810     case Int64.toInt( f( calT( K, NumAttr ), DomainAll ) ) of Index => (
811     case Array.sub( MappingLst, Index ) of R as ( I, V ) => (
812   case ( Index < List.length( LstAttrDesc ) ) of
813       true => nominalSplit( I, 0 )
814     | false => continuousSplit( I, V )
815     )
816     )
817   end
818
819 fun findMajorClassHelper (dataNil, Lst) = Lst
820 | findMajorClassHelper (dataCons(trainingInstance(Instance, Cl), Data), Lst
      ) =
821     let
822       fun updateListClass (nil, C) = [(C, 1)]
823       |  updateListClass ((IndexClass, Count)::T, C) =
824            case (IndexClass = C) of
825              true => (IndexClass, Count + 1 ) :: T
826            | false => (IndexClass, Count) :: updateListClass(T, C)
827     in findMajorClassHelper (Data, updateListClass(Lst, Cl))
828     end
829
830
831 fun findMajorClass (Data) =
832   let
833     fun findMaxClass (nil, MaxCount, MaxClass) = MaxClass
834     | findMaxClass ((Cl, Count) :: T, MaxCount, MaxClass) =
835       case (Count > MaxCount) of
836         true => findMaxClass(T, Count, Cl)
837       | false => findMaxClass (T, MaxCount, MaxClass)
838   in findMaxClass (findMajorClassHelper(Data, nil), ~1, class ~1)
839   end
840
841 fun checkSameClass (dataNil, CurClass) = true
842 |   checkSameClass (dataCons(trainingInstance(Instance, Cl), Data),
      CurClass) =
843   case (Cl = CurClass) of
844     true => checkSameClass(Data, CurClass)
845   | false => false
```

```
846
847 fun isLoop ( LstData : data list ) : bool =
848   case LstData of
849     nil => false
850   | H :: T => (
851   case H of
852     dataNil => true
853   | _ => isLoop ( T )
854   )
855
856 fun makeNumRand( Num ) = Real.fromInt ( floor (log2(Num) + 1.0) );
857
858 fun buildTree (Data0, LstAttrDesc, NumClass, NumAttr, K, R, AllNom ) =
859   case Data0 of
860   dataNil => leaf (class ~1)
861   | Data2 as dataCons(trainingInstance(Instance, Cl), Data) => (
862   if ( LstAttrDesc = nil) andalso AllNom
863   then
864     leaf (findMajorClass (Data))
865   else
866     let
867       fun buildTreeList(Lst, LstAttrDesc, NumClass, NumAttr, K, R) =
868         case Lst of
869           nil => treeListNil
870         | (H :: T) =>
871             treeListCons(
872               buildTree(H, LstAttrDesc, NumClass, NumAttr, K, R, AllNom),
873               buildTreeList(T, LstAttrDesc, NumClass, NumAttr, K, R))
874     in
875       case checkSameClass (Data2, Cl) of
876         true => leaf Cl
877       | false => (
878       case findBestSplit( Data2, LstAttrDesc, NumClass, NumAttr, K, R ) of
879         nominalSplit(~1, _) => leaf (class ~1)
880       | continuousSplit(~1, _) =>
881           leaf (findMajorClass(dataCons(trainingInstance(Instance, Cl),
    Data)))
882       (* leaf (class ~1) *)
883       | Sp as nominalSplit (_, _) => dn(
884                 Sp,
885                 buildTreeList(
886                   splitData( Data2, Sp, LstAttrDesc ),
887                   updateListAttr(Sp, LstAttrDesc),
888                   NumClass, NumAttr   1.0, K, R
889                 )
890               )
891       | Sp as continuousSplit (_, _) =>
892         let
893           val LstData = splitData( Data2, Sp, LstAttrDesc )
894         in
895           case isLoop (LstData) of
896             true => leaf (class ~1)
897          | false => dn(
898                Sp,
899                buildTreeList(
900                  LstData,
901                  updateListAttr(Sp, LstAttrDesc),
902                  NumClass, NumAttr, K, R
```

```
903                 )
904               )
905       end
906       )
907       end
908   )
909
910 (* END: build tree *)
911
912 (* START: read instances *)
913
914 (*fun readInstance( Lst : real list) : instance =
915   case Lst of
916     nil => instanceNil
917   | H :: T => instanceCons( nominal (round H), readInstance( T ) )
918
919 fun readInstance(Lst) =
920   case Lst of
921     nil => instanceNil
922   | H :: T => instanceCons(continuous H, readInstance(T))*)
923
924 fun readInstance( Lst, LstHelper ) =
925   case Lst of
926     nil => instanceNil
927   | H :: T => (
928   case LstHelper of
929     "Nom" :: Tail => instanceCons( nominal (round H ), readInstance( T,
      Tail ) )
930   | "Num" :: Tail => instanceCons( continuous H, readInstance( T, Tail ) )
931   )
932
933 fun readtrainingInstance(Lst, Cl, LstHelper) =
934       trainingInstance(readInstance(Lst, LstHelper), class Cl);
935
936 fun readDataToStructure(LstData, LstClass, LstHelper) = case LstData of
937   nil => dataNil
938 | H :: T => (
939     case LstClass of
940       HCl::TCl => dataCons(readtrainingInstance(H, HCl, LstHelper),
      readDataToStructure(T, TCl, LstHelper))
941   )
942
943 exception lengthNotMatch_readDataLstToStructure
944 fun readDataLstToStructure(LstLstData, LstLstClass, LstHelper) =
945   case (length(LstLstData) = length(LstLstClass)) of
946     false => raise lengthNotMatch_readDataLstToStructure
947   | true => (
948   case LstLstClass of
949     nil => nil
950   | HCl::TCl => (
951   case LstLstData of
952     H::T =>
953       readDataToStructure(H, (List.map round HCl), LstHelper)::
      readDataLstToStructure(T, TCl, LstHelper)
954     )
955  )
956
957 fun readListInstance(Lst, LstHelper) = case Lst of
```

```
  nil => nil
| H::T => readInstance(H, LstHelper)::readListInstance(T, LstHelper)

(* END: read instances *)


(* START: Build Forest  *)

fun createListRand(Num, Count, Lst, R) = case Count of
  0 => Lst
| K => (
    case Random.randRange(0, Num 1) of
      NextInt => createListRand(Num, K 1, (NextInt R)::Lst, R)
 )

fun createListListRand(NumList, Num) = case NumList of
  0 => nil
  | K => createListRand(Num, Num, nil, Random.rand(1, NumList))
           :: createListListRand( NumList   1, Num)

fun createOneRandData(LstData, LstRand) =
  case Array.fromList(LstData) of
    ArrData =>
  let
    fun createOneRandDataHelper(LstRand) =
      case LstRand of
        nil => nil
      | H :: T => Array.sub(ArrData, H) :: createOneRandDataHelper(T)
  in  createOneRandDataHelper(LstRand)
  end

fun createRandSets(LstData, LstLstRand) =
  case LstLstRand of
    nil => nil
  | H :: T => createOneRandData(LstData, H) :: createRandSets(LstData, T)

fun buildForestHelper(Sets, LstAttrDesc, NumClass, NumAttr, R, AllNom ) =
  case Sets of
    nil => nil
  | H :: T =>
    buildTree(H, LstAttrDesc, NumClass, NumAttr, makeNumRand ( NumAttr ), R
    , AllNom )
    :: buildForestHelper(T, LstAttrDesc, NumClass, NumAttr, R, AllNom)

exception noTree_createTrainingSets

fun buildForest(Train_data, Train_class : real list, NumTrees, LstAttrDesc,
     NumClass, NumAttr, R, LstHelper) =
  case createListListRand(NumTrees, length(Train_class)) of
    nil => raise noTree_createTrainingSets
  | lstLstRand =>
  let
    val AllNom = ( List.length( LstAttrDesc ) = round NumAttr )
  in
      buildForestHelper(
        readDataLstToStructure(createRandSets(Train_data, lstLstRand),
        createRandSets(Train_class, lstLstRand), LstHelper),
        LstAttrDesc, NumClass, NumAttr, R, AllNom)
```

```
1014   end
1015
1016 (* END: Build Forest *)
1017
1018
1019 (* START: TEST *)
1020 exception outOfBound_getTreeAtIndex;
1021 fun getTreeAtIndex (treeListNil, Index) = raise outOfBound_getTreeAtIndex
1022 |  getTreeAtIndex (treeListCons (Tree, TreeList), 0) = Tree
1023 |  getTreeAtIndex (treeListCons (Tree, TreeList), N) = getTreeAtIndex (
       TreeList, N   1 )
1024
1025 exception disagreeDataTypeAttrAndSplitPoint
1026 fun singleCaseTest (Ins, leaf L) = L
1027 |  singleCaseTest (Ins, dn (nominalSplit (AttrIndex, Unknown), TreeList)) =
1028   let
1029     val AttrValue =
1030       case getValueAttrAtIndex(Ins, AttrIndex) of
1031         nominal V => V
1032       | continuous V => raise disagreeDataTypeAttrAndSplitPoint
1033   in singleCaseTest (removeAttrFromInstance( Ins, AttrIndex) ,
       getTreeAtIndex(TreeList, AttrValue))
1034   end
1035 |  singleCaseTest (Ins, dn (continuousSplit (AttrIndex, Value), TreeList))
       =
1036   let
1037     val AttrValue =
1038       case getValueAttrAtIndex(Ins, AttrIndex) of
1039         nominal V => raise disagreeDataTypeAttrAndSplitPoint
1040       | continuous V => V
1041   in (
1042     case (AttrValue < Value) of
1043       true => singleCaseTest (Ins ,getTreeAtIndex(TreeList, 0))
1044     | false => singleCaseTest (Ins ,getTreeAtIndex(TreeList, 1))
1045   )
1046   end
1047
1048 exception sizeNotMatch_forestTestHelper;
1049 fun treeTestHelper(lstTestInst, lstClass, Tree) =
1050   case lstTestInst of
1051   nil => 0
1052 | h::t => (case lstClass of
1053       nil => raise sizeNotMatch_forestTestHelper
1054       | hCl::tCl => (case (singleCaseTest(h, Tree) = class hCl) of
1055         true => 1+treeTestHelper(t, tCl, Tree)
1056         | false => treeTestHelper(t, tCl, Tree)
1057       )
1058     );
1059
1060 fun treeTest(lstTestInst, lstClass, tree) =
1061   Real.fromInt(treeTestHelper(lstTestInst, lstClass, tree))/Real.fromInt(
       length(lstClass))*100.00;
1062 (* END: TEST *)
1063
1064 (* START: Forest test *)
1065
1066 fun forestSingleCaseTestHelper(Ins, LstTree) =
1067   case LstTree of
```

```
1068      nil => nil
1069    | H :: T => singleCaseTest(Ins, H) :: forestSingleCaseTestHelper(Ins, T)
1070
1071  fun updateVoting( Cl : class_value, LstCount) =
1072    case LstCount of
1073      nil => [(Cl, 1)]
1074    | (C, Num) :: T => (
1075        case ( C = Cl) of
1076          true => (Cl, Num + 1 ) :: T
1077        | false => (C, Num) :: updateVoting(Cl, T)
1078    )
1079
1080  fun votingHelper(LstVote, LstCount) =
1081    case LstVote of
1082      nil => LstCount
1083    | H :: T => votingHelper(T, updateVoting(H, LstCount))
1084
1085  fun voting(LstVote) =
1086    let
1087      fun findMaxClass (nil, MaxCount, MaxClass) = MaxClass
1088      | findMaxClass ((Cl, Count) :: T, MaxCount, MaxClass) =
1089          case (Count > MaxCount) of
1090            true => (
1091              case (Cl = class ~1) of
1092                true => findMaxClass (T, MaxCount, MaxClass)
1093              | false => findMaxClass(T, Count, Cl)
1094            )
1095          | false => findMaxClass (T, MaxCount, MaxClass)
1096    in findMaxClass(votingHelper(LstVote, nil), ~1, class ~1)
1097    end
1098
1099  fun forestSingleCaseTest(Ins, Forest) = voting(forestSingleCaseTestHelper(
        Ins, Forest))
1100
1101  exception sizeNotMatch_forestTestHelper
1102
1103  fun forestTestHelper(LstTestInst, LstClass, Forest) =
1104    case LstTestInst of
1105      nil => 0
1106    | H :: T => (
1107        case LstClass of
1108          nil => raise sizeNotMatch_forestTestHelper
1109        | HCl :: TCl => (
1110            case (forestSingleCaseTest(H, Forest) = class HCl) of
1111              true => 1 + forestTestHelper(T, TCl, Forest)
1112            | false => forestTestHelper(T, TCl, Forest)
1113        )
1114    )
1115
1116  fun forestTest(LstTestInst, LstClass, Forest) =
1117    Real.fromInt(forestTestHelper(LstTestInst, LstClass, Forest))/Real.
       fromInt(length(LstClass))*100.00
1118
1119  (* END: Forest test *)
1120
1121  (* START: read from files *)
1122
1123  fun getFirstCharList (Cli : char list) =
```

```
1124   case Cli of
1125     nil => nil
1126   | H::T => if H = #"," orelse H = #" " then nil else H :: getFirstCharList
       ( T )
1127
1128 fun parseLine (Line : char list) : real list =
1129   case Line of
1130     nil => nil
1131   | #"," :: Tail => parseLine( Tail )
1132   | #" " :: Tail => parseLine( Tail )
1133   | _ =>
1134     let
1135       val NumStr = implode( getFirstCharList Line)
1136       val Num = valOf ( Real.fromString( NumStr ) )
1137     in
1138       Num :: parseLine( List.drop( Line, size( NumStr ) ) )
1139     end
1140
1141 exception readData_ERRORINPUT
1142 fun readData( FileName, NLines) : real list list =
1143 let
1144   fun readLines ( Fh, NLines ) =
1145     case ( TextIO.endOfStream Fh, NLines = 0) of
1146       ( _ , true ) => ( TextIO.closeIn Fh; [])
1147     | ( false, false ) =>
1148     ( parseLine( explode( valOf ( TextIO.inputLine Fh ) ) ) ) :: readLines
      ( Fh, NLines   1)
1149     | ( true, false ) => raise readData_ERRORINPUT
1150 in
1151   readLines(TextIO.openIn FileName, NLines)
1152 end
1153
1154 exception readClass_ERRORINPUT
1155 fun readClass(FileName, NLines) : real list =
1156 let
1157   fun readLines (Fh, NLines) =
1158     case (TextIO.endOfStream Fh, NLines = 0) of
1159       ( _ , true ) => ( TextIO.closeIn Fh; [] )
1160     | ( false, false ) =>
1161     valOf( Real.fromString( implode( getFirstCharList( explode( valOf(
      TextIO.inputLine Fh ) ) ) ) ) )
1162       :: readLines ( Fh, NLines   1)
1163     | ( true,false ) => raise readClass_ERRORINPUT
1164 in
1165   readLines(TextIO.openIn FileName, NLines)
1166 end
1167
1168 (* Start: Run n folds *)
1169 fun checkContain (ls : int list, value : int) : bool =
1170 case ls of
1171  nil => false
1172  | h :: tl => if h = value then true else checkContain (tl, value)
1173
1174 fun rdomList (range : int, num: int, seed : Random.rand, resList) : int
      list =
1175 case num = 0 of
1176  true => resList
1177  | false => (
```

```
  case Random.randRange(0, range) seed of rdomVal =>
  case checkContain(resList, rdomVal) of
   true => rdomList (range, num, seed, resList)
   | false => rdomList(range, num   1, seed, rdomVal :: resList )
   )

fun trainValidSplit ( X : real list list, y : real list, validRowsList :
    int list, iter : int, X_train : real list list,
y_train : real list, X_valid : real list list, y_valid : real list) : real
    list list * real list * real list list  * real list =
case y of
 nil => (X_train, y_train, X_valid, y_valid)
 | h :: t => (
  case checkContain( validRowsList, iter) of
   true => trainValidSplit( tl X, tl y, validRowsList, iter + 1, X_train,
    y_train, (hd X)::X_valid, (hd y) :: y_valid)
   | false => trainValidSplit( tl X, tl y, validRowsList, iter + 1, (hd X)
    :: X_train, (hd y) :: y_train, X_valid, y_valid)
 )

fun nCVSplit(X : real list list, y : real list, test_ratio : real, numCV :
    int) : real list list list * real list list
 * real list list list * real list list =
let
 val numData = List.length y
 val numValid : int = Real.floor (test_ratio * Real.fromInt numData)
 val numTrain : int = numData   numValid
 val seed = Random.rand(2, 2015)
 fun oneCV (iter : int, X_train_list, y_train_list, X_valid_list,
    y_valid_list) =
 case iter = numCV + 1 of
  true => (X_train_list, y_train_list, X_valid_list, y_valid_list)
  | false => (
   let
    val validRowsList = rdomList(numData   1, numValid, seed, [])
    val (X_train, y_train, X_valid, y_valid) = trainValidSplit(X, y,
    validRowsList, 0, [], [], [], [])
   in
    oneCV(iter + 1, X_train ::X_train_list, y_train::y_train_list, X_valid
    ::X_valid_list, y_valid::y_valid_list)
   end
   )
in
 oneCV(1,[],[],[],[])
end

fun create_n_fold ( DT, Ratio, Folds, Train, Test ) = case DT of
  nil => (Train, Test)
| H as ( D, C, N, NL, NC, NA, HL ) :: T =>  (
  let
    val Data = readData(D, N)
    val Class = readClass(C, N)
    val ( LstDataTrain, LstClassTrain, LstDataTest, LstClassTest ) =
    nCVSplit(Data, Class, Ratio, Folds)
  in
    let
    fun create_n_fold_helper1 ( LstData, LstClass, Lst) =
      case LstData of
```

```
1227      nil => Lst
1228    | HD :: TD => (
1229        case LstClass of (HC :: TC) =>
1230          create_n_fold_helper1(TD, TC, Lst@[(HD, HC, N, NL, NC, NA, HL)])
1231      )
1232  in
1233    let
1234      fun create_n_fold_helper2 ( LstData, LstClass, Lst) =
1235        case LstData of
1236          nil => Lst
1237        | HD :: TD => (
1238        case LstClass of (HC :: TC) =>
1239          create_n_fold_helper2(TD, TC, Lst@[(HD, HC, N, HL)])
1240  )
1241    in
1242    let
1243      val Tr1 = create_n_fold_helper1( LstDataTrain, LstClassTrain, nil ) @
      Train
1244      val Te1 = create_n_fold_helper2( LstDataTest, LstClassTest, nil ) @
     Test
1245    in
1246      create_n_fold ( T, Ratio, Folds, Tr1, Te1 )
1247  end
1248  end
1249  end
1250  end
1251  )
1252
1253 (* END: Run n folds *)
1254
1255 fun main (
1256       ( NumTrees, TrainData, TrainClass, NumData, LstAttrDesc, NumClass,
     NumAttr, LstHelper )
1257       ) : tree list =
1258   buildForest(
1259     TrainData, TrainClass, NumTrees, LstAttrDesc, NumClass,
1260     NumAttr, Random.rand ( 1, 1 ), LstHelper
1261     )
1262
1263 val Datasets1 = [
1264
1265 (* 0 *)
1266 ("/local/RF/nursery_X_12960.csv",
1267 "/local/RF/nursery_y_12960.csv",
1268 12960, [3, 5, 4, 4, 3, 2, 3, 3], 5, 8.0,
1269 ["Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom"]
1270 ),
1271
1272 (* 1 *)
1273 ("/local/RF/marketing_X_6876.csv",
1274 "/local/RF/marketing_y_6876.csv",
1275 6876, [2, 5, 7, 6, 9, 5, 3, 9, 10, 3, 5, 8, 3], 9, 13.0,
1276 ["Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom
     ", "Nom", "Nom"]
1277 ),
1278
1279 (* 2 *)
1280 ("/local/RF/tic tac toe_X_958.csv",
```

```
"/local/RF/tic tac toe_y_958.csv",
958, [3,3,3,3,3,3,3,3,3], 2, 9.0,
["Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom"]
),

(* 4 *)
("/local/RF/kr vs k_X_28056.csv",
"/local/RF/kr vs k_y_28056.csv",
28056, [8,8,8,8,8,8], 18, 6.0,
["Nom", "Nom","Nom", "Nom","Nom", "Nom"]
),

(* 5 *)
("/local/RF/contraceptive_X_1473.csv",
"/local/RF/contraceptive_y_1473.csv",
1473, [34,4,4,17,2,2,4,4,2], 3, 9.0,
["Nom", "Nom","Nom", "Nom","Nom", "Nom", "Nom","Nom", "Nom"]
),


(* 7 *)
("/local/RF/vehicle_X_846.csv",
"/local/RF/vehicle_y_846.csv",
846, [], 4, 18.0,
["Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num
    ", "Num", "Num", "Num", "Num", "Num", "Num"]
),

(* 19 *)
("/local/RF/winequality red_X_1599.csv",
"/local/RF/winequality red_y_1599.csv",
1599, [], 11, 11.0,
["Num", "Num","Num", "Num","Num", "Num","Num", "Num","Num", "Num","Num"]
)

]


val Datasets2 = [

(* 8 *)
("/local/RF/banana_X_5300.csv",
"/local/RF/banana_y_5300.csv",
5300, [], 2, 2.0,
["Num", "Num"]
),

(* 10 *)
("/local/RF/soybean_X_683.csv",
"/local/RF/soybean_y_683.csv",
683, [7, 2, 3, 3, 2, 4, 4, 3, 3, 3, 2, 2, 3, 3, 3, 2, 2, 3, 2, 2, 4, 4, 2,
    3, 2, 3, 2, 4, 5, 2, 2, 2, 2, 2, 3], 19, 35.0,
["Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom
    ", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom",
     "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "
    Nom", "Nom", "Nom", "Nom"]
),

```

```
(* 11 *)
("/local/RF/chess_X_3196.csv",
"/local/RF/chess_y_3196.csv",
3196, [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2,
     2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ,2, 2, 2], 2, 36.0,
["Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom
    ", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom",
     "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "Nom", "
    Nom", "Nom", "Nom", "Nom", "Nom"]
),

(* 12 *)
("/local/RF/splice_X_3190.csv",
"/local/RF/splice_y_3190.csv",
3190,
    [5,5,4,4,4,4,4,4,4,4,4,4,4,5,4,4,4,4,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,6,6,5,5,5,5,5,5,
     3, 60.0,
["Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "
    Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "
    Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "
    Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "
    Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "Nom","Nom", "
    Nom","Nom", "Nom","Nom", "Nom"]
),

(* 13 *)
("/local/RF/penbased_X_10992.csv",
"/local/RF/penbased_y_10992.csv",
10992, [], 10, 16.0,
["Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num
    ", "Num", "Num", "Num", "Num", "Num"]
),

(* 15 *)
("/local/RF/phoneme_X_5404.csv",
"/local/RF/phoneme_y_5404.csv",
5404, [], 2, 5.0,
["Num", "Num", "Num", "Num", "Num"]
),

(* 17 *)
("/local/RF/abalone_X_4177.csv",
"/local/RF/abalone_y_4177.csv",
4177, [3], 29, 8.0,
["Nom","Num", "Num","Num", "Num","Num", "Num","Num"]
),

(* 18 *)
("/local/RF/page blocks_X_5472.csv",
"/local/RF/page blocks_y_5472.csv",
5472, [], 5, 10.0,
["Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num"]
),


(* 20 *)
("/local/RF/winequality white_X_4898.csv",
"/local/RF/winequality white_y_4898.csv",
```

```
4898, [], 11, 11.0,
["Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num", "Num"]
),

(* 21 *)
("/local/RF/CTG_X_2126.csv",
"/local/RF/CTG_y_2126.csv",
2126, [], 11, 21.0,
["Num","Num", "Num","Num", "Num","Num", "Num","Num","Num","Num", "Num","Num", "Num","Num", "Num","Num","Num", "Num","Num", "Num","Num"]
),

(* 22 *)
("/local/RF/satimage_X_6435.csv",
"/local/RF/satimage_y_6435.csv",
6435, [], 7, 36.0,
["Num","Num", "Num","Num", "Num","Num", "Num","Num","Num","Num", "Num","Num", "Num","Num", "Num","Num","Num", "Num","Num", "Num","Num", "Num","Num","Num", "Num","Num", "Num","Num", "Num","Num","Num","Num", "Num","Num"]
),

(* 24 *)
("/local/RF/jsbach_chorals_harmony_X_5665.csv",
"/local/RF/jsbach_chorals_harmony_y_5665.csv",
5665, [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 16, 5], 102, 14.0,
["Nom","Nom","Nom","Nom","Nom","Nom","Nom","Nom","Nom","Nom","Nom","Nom","Nom"]
)

]


val Seed1 = 719561528
val Seed2 = 937436219

val RandState = Random.rand( Seed1, Seed2 )
val randInt = fn() => Random.randInt (RandState)
val randNat = fn() => Random.randNat (RandState)
val randReal = fn() => Random.randReal (RandState)
val randRange = fn(Low,High) => Random.randRange (Low,High) (RandState)


fun addNumTrees( TrainData, TrainClass, NumData, LstAttrDesc, NumClass, NumAttr, LstHelper ) =
  ( randRange( 10, 30 ), TrainData, TrainClass, NumData, LstAttrDesc, NumClass, NumAttr, LstHelper )


val ( Inputs', Outputs ) = create_n_fold ( Datasets1, 0.3, 5, nil, nil )
val Inputs' = rev Inputs'
val Outputs = rev Outputs

val Inputs = map( addNumTrees, Inputs' )

val ( Test_inputs', Validation_outputs ) = create_n_fold ( Datasets2, 0.3, 5, nil, nil )
```

```
1429 val Test_inputs' = rev Test_inputs'
1430 val Validation_outputs = rev Validation_outputs
1431
1432 val Test_inputs = map( addNumTrees, Test_inputs' )
1433
1434 val All_outputs = Outputs @ Validation_outputs
1435
1436 val Funs_to_use = [
1437   "rNil", "rCons",
1438   "splitNil", "splitCons",
1439   "domainNil", "domainCons",
1440   "evalNil", "evalCons",
1441   "false", "true",
1442   "realLess", "realAdd", "realSubtract", "realMultiply",
1443   "realDivide", "tanh",
1444   "tor", "rconstLess",
1445   "ln"
1446
1447 ]
1448
1449
1450 val Abstract_types = [ ]
1451 val Reject_funs = []
1452 fun restore_transform D = D
1453 fun compile_transform D = D
1454 val print_synted_program  = Print.print_dec'
1455
1456 val AllAtOnce = false
1457
1458
1459 exception MaxSyntComplExn
1460 val MaxSyntCompl = (
1461   case getCommandOption "  maxSyntacticComplexity" of
1462     NONE => 500.0
1463   | SOME S => case Real.fromString S of SOME N => N
1464   ) handle Ex => raise MaxSyntComplExn
1465
1466
1467
1468 val OnlyCountCalls = false
1469 val TimeLimit : Int.int = 500000000
1470 val max_time_limit = fn () => Word64.fromInt TimeLimit : Word64.word
1471 val max_test_time_limit = fn () => 0w1000000000 : Word64.word
1472 val time_limit_base = fn () => real TimeLimit
1473
1474 fun max_syntactic_complexity() = MaxSyntCompl
1475 fun min_syntactic_complexity() = 0.0
1476 val Use_test_data_for_max_syntactic_complexity = false
1477
1478 val main_range_eq = op=
1479 val File_name_extension = ""
1480 val Resolution = NONE
1481 val StochasticMode = false
1482
1483 val Number_of_output_attributes : Int64.int = 4
1484
1485 fun to ( G : real ) : LargeInt.int =
1486  Real.toLargeInt IEEEReal.TO_NEAREST(G * 1.0e14 )
```

```
1487
1488 structure Grade : GRADE =
1489 struct
1490 type grade = LargeInt.int
1491 val NONE = LargeInt.maxInt (* To check that LargeInt has infinite precision
     . *)
1492 val zero = LargeInt.fromInt 0
1493 val op+ = LargeInt.+
1494 val comparisons = [ LargeInt.compare ]
1495 fun toString( G : grade ) : string =
1496   Real.toString( Real.fromLargeInt G / 1.0E14 )
1497 val N = LargeInt.fromInt 1000000 * LargeInt.fromInt 1000000
1498 val significantComparisons = [ fn ( E1 , E2 )
1499   => LargeInt.compare ( E1 div N, E2 div N ) ]
1500 fun toString ( G : grade ) : string =
1501   Real.toString ( Real.fromLargeInt G / 1.0E14 )
1502 val pack = LargeInt.toString
1503 fun unpack( S : string ) : grade =
1504   case LargeInt.fromString S of SOME G => G
1505
1506 val post_process = fn X => X
1507 val toRealOpt = NONE
1508 end
1509
1510
1511 fun output_eval_fun( exactlyOne( I : Int.int, _ , lstTrees: main_range  ) )
1512     : { numCorrect : Int.int, numWrong : Int.int, grade : Grade.grade }
      List.list = [
1513 let
1514   val (TestData, TestClass, NumData, LstHelper) = List.nth(All_outputs, I)
1515   val error = 100.0   forestTest((readListInstance(TestData, LstHelper)), (
      List.map round TestClass), lstTrees);
1516 in
1517   if Real.==( error, 0.0 ) then
1518     { numCorrect = 1, numWrong = 0, grade = to error}
1519   else if error > 1.0E30 orelse not( Real.isNormal error ) then
1520    { numCorrect = 0, numWrong = 1, grade = to 1.0E30}
1521   else
1522    { numCorrect = 1, numWrong = 0, grade = to error}
1523 end
1524 ]
1525
1526
1527 type int = Int64.int
1528 type word = Word64.word
```

Listing A.2: Specification file for the Construction of Classifiers experiment

Appendix B

Improved programs

B.1 Experiment 1 - The combination of classifiers experiment

Optimized program

```
fun f TupleList =
  let
    fun updateVoting( Cl as class( I1 ), LstCount ) =
      case LstCount of
        classCountListNil =
          classCountListCons(
            classCount( Cl, 1.0 ),
            classCountListNil
            )
      | classCountListCons(
          CC as classCount( C as class( CI ), Num ),
          Tail
          ) =
      case classEq( C, Cl ) of
        false = Tail
      | true = classCountListCons( CC, LstCount )
  in
    let
      fun votingHelper TupleLst =
        case TupleLst of
          tupleListNil = classCountListNil
        | tupleListCons( Tu as tuple( H as class( HI ), L ), T ) =
            updateVoting( H, votingHelper( T ) )
    in
      case votingHelper( TupleList ) of
        classCountListNil = (
          case TupleList of
            tupleListNil = (raise NA_C175D)
          | tupleListCons(
              VC175E as tuple( VC175F as class( VC1760 ), VC1761 ),
              VC1762
              ) =
              VC175F
          )
      | classCountListCons(
          VC1763 as classCount( VC1764 as class( VC1765 ), VC1766 ),
```

```
            VC1767
            ) =
            VC1764
    end
  end
```

Listing B.1: Result for the Combination of Classifiers experiment - The optimized program

B.2 Experiment 2 - The construction of classifiers experiment

Improved program number 1

```
fun f( T, Ds ) =
  let
    fun evals Ds' =
      let
        fun eval( I, Splits, Sum, OrderNumber ) =
          let
            fun newEntropy( Cards, CardSum, CardRand ) =
              case Cards of
                rNil => T
              | rCons( Card1, Cards1 ) =>
                  realSubtract(
                    newEntropy( Cards1, CardSum, CardRand ),
                    realMultiply(
                      realDivide( Card1, CardSum ),
                      ln( realDivide( Card1, CardSum ) )
                      )
                    )
          in
            case Splits of
              splitNil => 0.0
            | splitCons(
                Split1 as ( Cards', CardSum', CardRand' ),
                Splits1
                ) =>
                realAdd(
                  realMultiply(
                    realDivide( CardSum', Sum ),
                    newEntropy( Split1 )
                    ),
                  eval( I, Splits1, Sum, OrderNumber )
                  )
          end
      in
        case Ds' of
          domainNil => evalNil
        | domainCons(
            D1' as ( I1, Splits1', Sum1, OrderNumber1 ),
            Ds1
            ) =>
            evalCons(
              ( I1, Splits1', Sum1, OrderNumber1, eval( D1' ) ),
              evals( Ds1 )
              )
      end
  in
```

```
    let
      fun filter Es2 =
        case Es2 of
          evalNil => evalNil
        | evalCons(
            E3 as ( I3, Splits3, Sum3, OrderNumber3, Eval3 ),
            Es3
            ) =>
        case realLess( OrderNumber3, T ) of
          false => filter( Es3 )
        | true => evalCons( E3, filter( Es3 ) )
    in
      let
        fun min Es =
          case Es of
            evalNil => (raise NA1)
          | evalCons(
              E4 as ( I4, Splits4, Sum4, OrderNumber4, Eval4 ),
              Es4
              ) =>
          case Es4 of
            evalNil => E4
          | evalCons(
              E5 as ( I5, Splits5, Sum5, OrderNumber5, Eval5 ),
              Es5
              ) =>
          case min( Es4 ) of
            E6 as ( I6, Splits6, Sum6, OrderNumber6, Eval6 ) =>
          case realLess( Eval4, Eval6 ) of false => E6 | true => E4
      in
        case min( filter( evals( Ds ) ) ) of
          E7 as ( I7, Splits7, Sum7, OrderNumber7, Eval7 ) => I7
      end
    end
  end
```

Listing B.2: Result for the Construction of Classifiers experiment - The improved program number 1

Improved program number 2

```
fun f( T, Ds ) =
  let
    fun evals Ds' =
      let
        fun eval( I, Splits, Sum, OrderNumber ) =
          let
            fun newEntropy( Cards, CardSum, CardRand ) =
              case Cards of
                rNil => CardRand
              | rCons( Card1, Cards1 ) =>
                  realSubtract(
                    newEntropy( Cards1, CardSum, CardRand ),
                    realMultiply(
                      realDivide( Card1, CardSum ),
                      ln( realDivide( Card1, CardSum ) )
                      )
                    )
          in
```

```
            case Splits of
              splitNil => 0.0
            | splitCons(
                Split1 as ( Cards', CardSum', CardRand' ),
                Splits1
                ) =>
                realAdd(
                  realMultiply(
                    realDivide( CardSum', Sum ),
                    newEntropy( Split1 )
                    ),
                  eval( I, Splits1, Sum, OrderNumber )
                  )
          end
      in
        case Ds' of
          domainNil => evalNil
        | domainCons(
            D1' as ( I1, Splits1', Sum1, OrderNumber1 ),
            Ds1
            ) =>
            evalCons(
              ( I1, Splits1', Sum1, OrderNumber1, eval( D1' ) ),
              evals( Ds1 )
              )
      end
  in
  let
        fun min Es =
          case Es of
            evalNil => (raise NA1)
          | evalCons(
              E4 as ( I4, Splits4, Sum4, OrderNumber4, Eval4 ),
              Es4
              ) =>
          case Es4 of
            evalNil => E4
          | evalCons(
              E5 as ( I5, Splits5, Sum5, OrderNumber5, Eval5 ),
              Es5
              ) =>
          case min( Es4 ) of
            E6 as ( I6, Splits6, Sum6, OrderNumber6, Eval6 ) =>
          case realLess( Eval4, Eval6 ) of false => E6 | true => E4
      in
        case min( evals( Ds ) ) of
          E7 as ( I7, Splits7, Sum7, OrderNumber7, Eval7 ) => I7
      end
  end
```

Listing B.3: Result for the Construction of Classifiers experiment - The improved program number 2

Improved program number 3

```
fun f( T, Ds ) =
  let
    fun evals Ds' =
      let
```

```
5         fun eval( I, Splits, Sum, OrderNumber ) =
6           let
7             fun newEntropy( Cards, CardSum, CardRand ) =
8               case Cards of
9                 rNil =>
10                  realMultiply(
11                    tanh(
12                      tanh(
13                        tor( rconst( 0, 0.25, ~0.472542806823 ) )
14                        )
15                      ),
16                    CardRand
17                    )
18                | rCons( Card1, Cards1 ) =>
19                    realSubtract(
20                      newEntropy( Cards1, CardSum, CardRand ),
21                      realMultiply(
22                        realDivide( Card1, CardSum ),
23                        realDivide( Card1, CardSum )
24                        )
25                      )
26            in
27              case Splits of
28                splitNil => tor( rconst( 0, 0.25, ~0.419265635596 ) )
29              | splitCons(
30                  Split1 as ( Cards', CardSum', CardRand' ),
31                  Splits1
32                  ) =>
33                  realAdd(
34                    realMultiply(
35                      realDivide( CardSum', Sum ),
36                      newEntropy( Split1 )
37                      ),
38                    eval( I, Splits1, Sum, OrderNumber )
39                    )
40          end
41      in
42        case Ds' of
43          domainNil => evalNil
44        | domainCons(
45            D1' as ( I1, Splits1', Sum1, OrderNumber1 ),
46            Ds1
47            ) =>
48            evalCons(
49              (
50                I1,
51                Splits1',
52                Sum1,
53                OrderNumber1,
54                tanh( tanh( eval( D1' ) ) )
55                ),
56              evals( Ds1 )
57              )
58      end
59  in
60      let
61        fun min Es =
62          case Es of
```

```
            evalNil => (raise NA1)
          | evalCons(
              E4 as ( I4, Splits4, Sum4, OrderNumber4, Eval4 ),
              Es4
              ) =>
          case Es4 of
            evalNil => E4
          | evalCons(
              E5 as ( I5, Splits5, Sum5, OrderNumber5, Eval5 ),
              Es5
              ) =>
          case min( Es4 ) of
            E6 as ( I6, Splits6, Sum6, OrderNumber6, Eval6 ) =>
          case realLess( Eval4, Eval6 ) of false => E6 | true => E4
      in
        case min( evals( Ds ) ) of
          E7 as ( I7, Splits7, Sum7, OrderNumber7, Eval7 ) => I7
      end
  end
```

Listing B.4: Result for the Construction of Classifiers experiment - The improved program number 3

Improved program number 4

```
fun f( T, Ds ) =
  let
    fun evals Ds' =
      let
        fun eval( I, Splits, Sum, OrderNumber ) =
          let
            fun newEntropy( Cards, CardSum, CardRand ) =
              case Cards of
                rNil =>
                  realMultiply(
                    tanh(
                      tanh(
                        tor(
                          rconst( 2, 0.25250625, ~0.470030306823 )
                          )
                        )
                      ),
                    CardRand
                    )
              | rCons( Card1, Cards1 ) =>
                  realSubtract(
                    newEntropy( Cards1, CardSum, CardRand ),
                    realMultiply(
                      realDivide( Card1, CardSum ),
                      realDivide( Card1, CardSum )
                      )
                    )
          in
            case Splits of
              splitNil => tor( rconst( 0, 0.25, ~0.419265635596 ) )
            | splitCons(
                Split1 as ( Cards', CardSum', CardRand' ),
                Splits1
                ) =>
```

```
                realAdd(
                  realMultiply(
                    realDivide( CardSum', Sum ),
                    newEntropy( Split1 )
                    ),
                  eval( I, Splits1, Sum, OrderNumber )
                  )
          end
      in
        case Ds' of
          domainNil => evalNil
        | domainCons(
            D1' as ( I1, Splits1', Sum1, OrderNumber1 ),
            Ds1
            ) =>
            evalCons(
              (
                I1,
                Splits1',
                Sum1,
                OrderNumber1,
                tanh( tanh( eval( D1' ) ) )
                ),
              evals( Ds1 )
              )
      end
  in
      let
        fun min Es =
          case Es of
            evalNil => (raise (raise (raise (raise (raise NA1)))))
          | evalCons(
              E4 as ( I4, Splits4, Sum4, OrderNumber4, Eval4 ),
              Es4
              ) =>
          case Es4 of
            evalNil => E4
          | evalCons(
              E5 as ( I5, Splits5, Sum5, OrderNumber5, Eval5 ),
              Es5
              ) =>
          case min( Es4 ) of
            E6 as ( I6, Splits6, Sum6, OrderNumber6, Eval6 ) =>
          case realLess( Eval4, Eval6 ) of false => E6 | true => E4
      in
        case min( evals( Ds ) ) of
          E7 as ( I7, Splits7, Sum7, OrderNumber7, Eval7 ) => I7
      end
    end
  end
```

Listing B.5: Result for the Construction of Classifiers experiment - The improved program number 4