Improving M5 Model Tree by Evolutionary Algorithm

Master's Thesis in Computer Science

Hieu Chi Huynh

May 15, 2015
Halden, Norway

Abstract

Decision trees are potentially powerful predictors, most well-known for their accuracy and the ability of explicitly representing the structure of datasets. The first part of this thesis can be viewed as a brief summary of decision tree methods, which covers all ideas and approaches employed in both classification and regression trees. Then we particularly focus on studying M5 model tree and its related models. M5 tree currently is state-of-the-art model among decision trees for regression task. Besides accuracy, it can take tasks with very high dimension - up to hundreds of attributes. Based on the understanding of M5 tree algorithm, our main purpose in this project is to improve M5 tree using Automatic programming and Evolutionary algorithm. We design and conduct experiments to investigate possibilities of altering pruning and smoothing part in M5 tree by programs synthesized by ADATE (Automatic Design of Algorithms Through Evolution) system. A short introduction and analysis of ADATE's power is also provided.

The experimental results show that we successfully improve M5 tree learners by Evolutionary algorithm. Further more, we overcome overfitting problem and make alternative programs generalized well to other datasets.

Acknowledgments

First and foremost, I would like to deeply thank my supervisor, Associate Professor Roland Olsson. Along the way of doing this thesis, he always patiently gives me advice and encouragement. Reports and discussion sessions with him every week have been motivating me a lot. Experiments in this thesis uses programs generated by ADATE system, which is invented and developed by Roland.

I am grateful to my family for being supportive to me whatever I do and whenever I need. Their care and encouragement ease my homesick.

I also would like to thank my best friend, Que Tran. Her countless help in life keeps me away from the loneliness and sometimes the hunger. She is always willing to discuss whatever topic I need and most of time I learned from those discussions.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background and motivation

The research topic in this dissertation is Machine Learning. So it is essential to clarify first, what is Machine Learning? According to Arthur Samuel (1959), Machine Learning is *"a field of study that gives computers the ability to learn without being explicitly programmed"*. A more formal definition was provided by Tom M. Mitchell [32], that is, *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E"*. Machine learning discovers the construction and study of algorithms that can analysis, learn and make predictions on data. It shares several concepts and is closely related to other fields as computational statistic, mathematical optimization, pattern recognition and artificial intelligent.

Generally, to solve a specific problem by machine learning, we need to have data describing that problem and a machine learning model. Data is split into training data and testing data. Machine learning model is constructed based on training data and expected to capture as most as possible regularities or patterns existed in data. Performance of the model is measured on testing data whose part is not used in training. Roughly say, there are two types of learning, *supervised* and *unsupervised* learning. Term supervised learning refers to problems, or data, that contain input and desired output, the machine learning model needs to build a mapping to produce outputs from given inputs. On the other hands, in unsupervised learning, there is no output data and the model needs to discover relationships between features themselves. This thesis focuses on two well-known supervised learning algorithms, Decision trees with an emphasis on regression trees and Evolutionary algorithm.

One branch of supervised learning, empirical learning, is concerned with building or revising models in the light of large numbers of exemplary cases, taking into account typical problems such as missing data and noise. Many of these models are constructed as classifiers which typically try to stick a label on each example in the data set. Decision tree related algorithms are one of most used ones because of its efficient, robust and relative simplicity. Nevertheless, there are also other tasks that require learned models to predict a numeric value associated with a set of given attributes. For instance, one might want to predict sea level based on weather conditions in a day such as temperature, wind speed, etc. Some obvious approaches to deal with these kinds of task is to separate numeric values at the class attribute into categories, e.g $0-4$, $5-10$, etc. These attempts often fail, partly because algorithms for building decision trees cannot make use of the implicit ordering of such classes. Instead, there are regression decision trees that are designed specially to handle these regression tasks. This project particularly investigates on M5 model tree, most well-known for its efficiency and robustness to high dimensional data.

More over, we typically aim to improve M5 model tree algorithm by Evolutionary Algorithm (EA). Inspired by biological natural selection process, EA continuously generates solution programs and keep ones that are better than current population according to a pre-determined criteria. Our motivations are that: (i) there are some parts in M5 tree algorithm not fully explained or reasoned about their goodness, thus it might be possible to have alternatives that offer higher performance; (ii) improve a well-known Machine learning model as M5 tree must be an interesting task; (iii) Evolutionary algorithm is a promising approach to use and there were projects actually successful in improving a Machine learning model by EA [17]. In our thesis, we employ ADATE (Automatic Design of Algorithms Through Evolution) system [2] as a EA tool to synthesize solutions for improvements of M5 tree.

1.2 Research question and method

1.2.1 Research question/Problem statement/Objectives

As mentioned, in this project we study decision trees, particularly focus on regression M5 model tree, and evolutionary algorithm. We intend to have two stages: first we gain knowledge and get a deeper insight into these topics; then comes to implementation stage, we apply ADATE system [2] to improve M5. The emphasis of this project will be on implementation stage where we hope to be able to get impressive results. However there are also two challenges need to be addressed in implementation stage. ADATE usually has to evaluate millions of program instances to find the best version, thus M5 tree instances need to run fast enough on selected datasets. Second, it is known that overfitting seems to be a problem for programs generated by ADATE and we need to learn how to prevent it from happening. The most important requirement is that found programs must apply successfully in other datasets.

Our purpose in this thesis is to find the answer for this research question:

RQ *How and to what extend can ADATE system improve M5 model tree?*

1.2.2 Method

The research question above could be broken into two sub-questions:

1. How ADATE can generate programs to change M5 algorithm and how much its performance can be improved?

2. Can the new M5 algorithm be generalized and effective on other datasets?

We approach these questions by studying the idea, theory and implementation behind related topics, that is, M5 tree in the manner of decision tree and ADATE system. Then based on knowledge got from first stage, we will attempt to answer these questions by analyzing and discussing experimental results. Important matters recognized before conducting experiments includes:

- Which part of a M5 tree will be improved?
- Which datasets will be used to train and evaluate programs' performance?
- Which criteria to determine the best program?
- Statistical methods for analyzing results?

1.3 Report Outline

The report of this thesis is presented into following chapters:

- Chapter 2 presents the importance and the role of decision tree models in Machine learning field, theoretically and practically.
- Chapter 3 gives an overview about decision tree. We cover sections as how to split nodes when growing trees, how to avoid overfitting by pruning, the difference between classification and regression trees, etc.
- Chapter 4 goes further into regression tree area with M5 model tree. We discuss methods employed in constructing a M5 tree learner and related models as M5' and M5Rules.
- Chapter 5 gives a brief presentation to concepts of Evolutionary algorithm in Machine Learning and Automatic programming. Then the ADATE system will be introduced as an implementation and founded based on these above concepts.
- Chapter 6 describes experiments in matters of design and implementation. We start by selecting targets, selecting datasets and dividing datasets into training and testing sets. Then we explain how we implement the design of experiments.
- Chapter 7 collects, analyzes and discusses the experimental results.
- Chapter 8 presents our conclusion about this project, as well as suggestions for future works.

Finally, we have Appendix sections with source code in implement stage of experiments.

Chapter 2

Role of Decision Trees in solving Machine Learning problems

Decision Trees are non-parametric supervised learning methods for classification and regression. The goal of such methods is to create tree-based models to predict values of target variables by learning simple decision rules from the data features. Generally, a decision tree can be viewed as a non-cyclic, directed graph where there is one root node, a set of interior nodes that correspond to one or several of input variables, and a set of leaf nodes that present numeric values of discrete labels given the values of the input variables represented by the path from the root to the leaf. At each interior node, decision tree divides a data set into subsets based on a test in a manner that reduces estimated errors as much as possible. Decision Trees method has been widely used by Machine learning community and commonly accepted as a powerful tool which proves "state of the art" in a number of Machine learning problems. The process of top-down induction of decision trees [39] is an example of a greedy algorithm, and it is by far the most common strategy for learning decision rules from data.

2.1 Advantages and applications of Decision Trees

Decision tree methods offer several advantages over other Machine learning algorithms:

- It is simple to understand and interpret the models. A decision tree can be converted to a collection of learning rules so that people who are not Machine learning practitioners could understand. Also most of trees can be visualized. Figure 2.1 shows an illustration of visualizing a decision tree learned for the Iris data set [7]. A decision tree model is also called a white box model, meaning that if a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.

- These methods do not take many pre-processing steps. Some other techniques, e.g linear regression, demands normalization, converting from categorized to numerical variables, or deleting blank values. Moreover, decision tree methods can handle both numeric and categorical data while other models often specialize in analysing datasets that have only one type of variable.

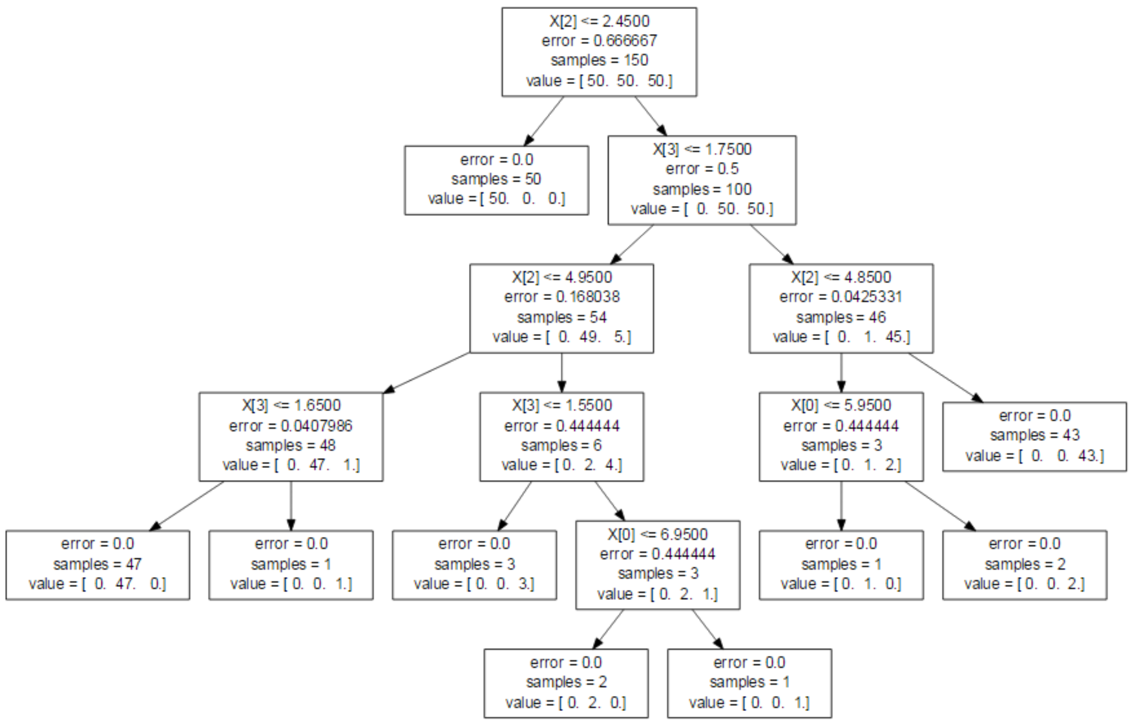


Figure 2.1: A decision tree built on Iris data set

- The cost of training a tree is low. Usually it is only logarithm of the number of data points, i.e instances, in dataset. Because of this, decision trees are often considered first for fast analysing a dataset to observe its nature and properties. The low training cost is also an advantage when it comes to handle large dataset.

- Decision trees can be used for multiple output problems.

- It is possible to validate a model using statistical test. That makes it possible to account for the reliability of the model.

  The oldest and most popular decision tree algorithm is ID3 (Iterative Dichotomiser 3), developed in 1986 by Ross Quinlann [39]. The algorithm creates a multi-way tree, finding for each node (i.e. in a greedy manner) the categorical feature that will yield the largest information gain for categorical targets. Trees are grown to their maximum size and then a pruning step is usually applied to improve the ability of the tree to generalise to unseen data. Later generation of decision tree methods are all based on Quinlan's approach,i.e growing the tree in a manner of most reducing the error and then pruning to avoid overfitting, but with modifications or improvements to increase the performance, e.g by better estimating errors on unseen cases. C4.5 is the successor to ID3 and removed the restriction that features must be categorical by dynamically defining a discrete attribute (based on numerical variables) that partitions the continuous attribute value into a discrete set of intervals. C4.5 converts the trained trees (i.e. the output of the ID3 algorithm) into sets of if-then rules. Furthermore, CART(Classification and Regression Tree) [12] and M5 model Tree

[44] support numerical target variables, their leaves contain linear models instead of labels as in classification problems.

To illustrate how a decision tree works on a classification task, Figure 2.2 visualizes decision boundaries made of each pair of features in Iris dataset. Decision boundaries are drawn by combing simple thresholding rules learned from training samples. A thresholding rule is usually a first-order comparison between value of a variable and a constant so edges forming decision boundaries tend to be parallel with axis.

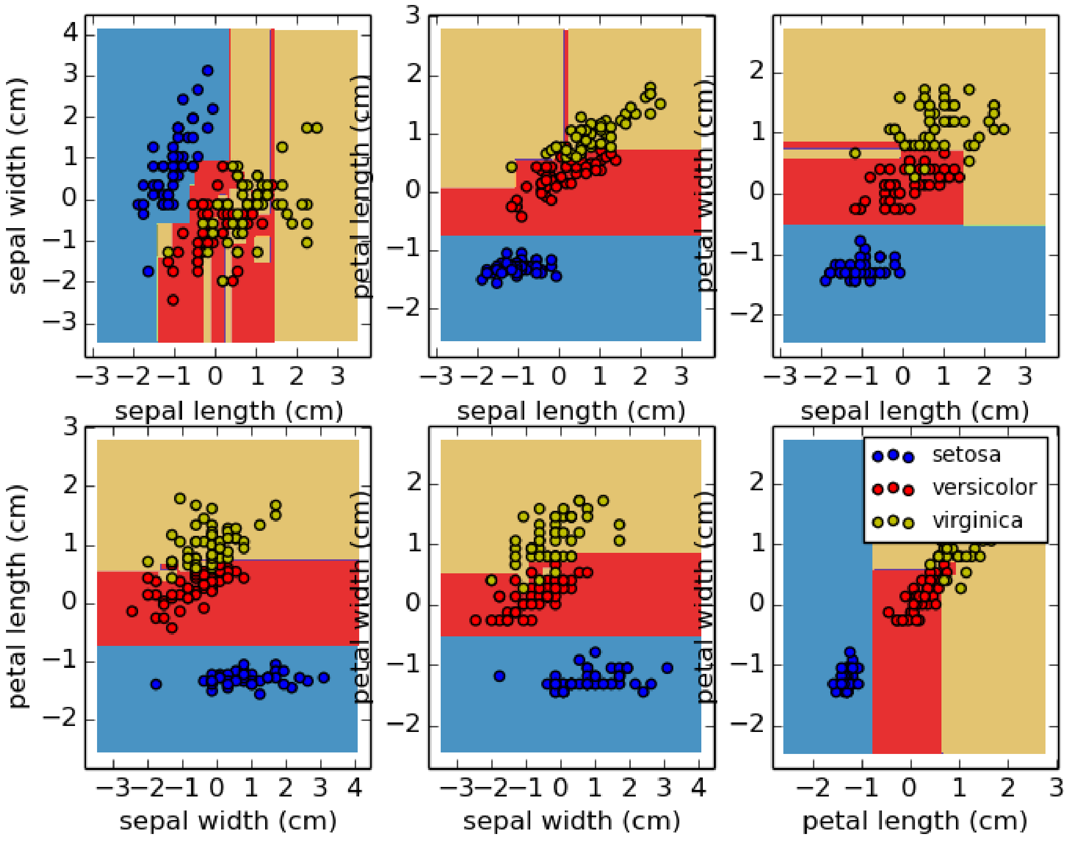


Figure 2.2: Decision surface of a decision tree using paired features on Iris

In terms of learning from a regression task, figure 2.3 presents estimated values of data points from a regression decision tree after fitting a Sine function and some noise. As shown in this figure, the decision line includes segments paralleling with two axis and it learns local linear regressions approximating the sine curve. We also can see that if the maximum depth of the tree (graph in red) is set too high, the decision trees learn too fine details of the training data and learn from the noise, i.e. they overfit.

As aforementioned, decision trees can handle multiple-output problems. This is the kind of problem where there are many target variables to be predicted. If target variables are independent to each other, a very simple way is to construct n different models, each for one of variables. However, because it is likely that the output values related to the same input are themselves correlated, an often better way is to build a single model capable of

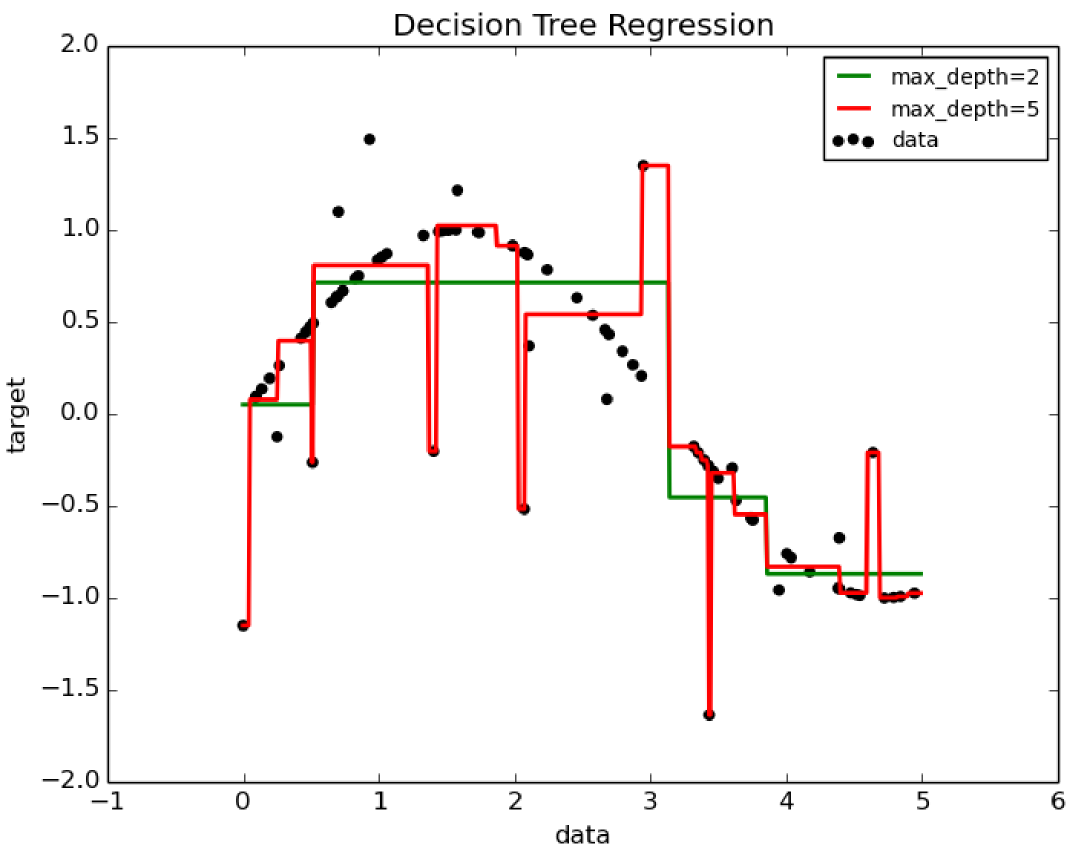


Figure 2.3: Decision Tree regression to learn Sine function

predicting simultaneously all n outputs. First, it requires lower training time since only a single estimator is built. Second, the generalization accuracy of the resulting estimator may often be increased. In [6], they attempted to exploit this property of decision tree to build a model for face completion task, that is, to predict the lower half of faces given their upper faces. The results got from multiple-output decision trees is presented along with ones from other models for a comparison (figure 2.4).

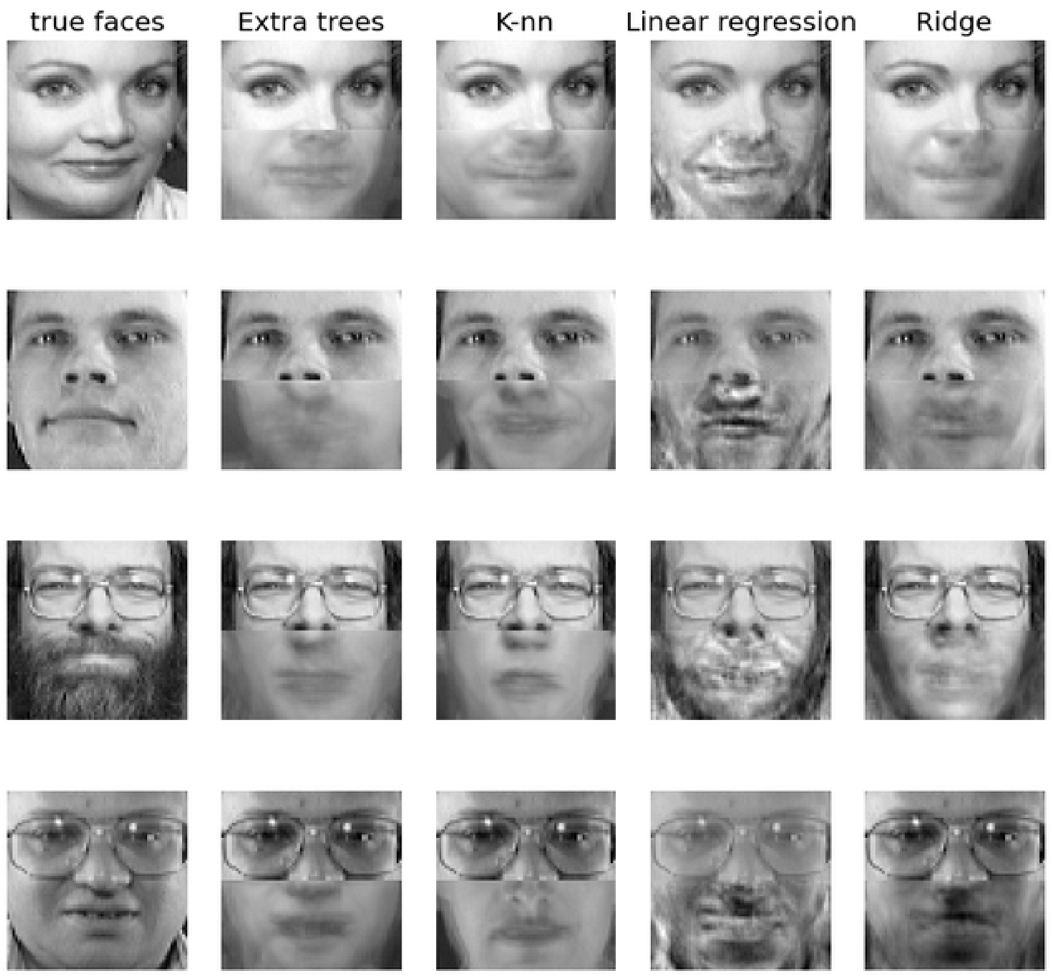


Figure 2.4: Face completion with a multi-output estimators

Another common use of decision tree is feature selection. At each internal node there is a feature selected to split the data in the manner that this feature offers the most reduction in estimated errors, and thus this feature can be considered as the most current informative feature. It is very often that a decision tree only uses a subset of most informative features for tree constructing. For datasets with very many attributes, one of method to reduce number of dimensions is to simply use subset of features exported by a decision tree model. In $C5.0$ software [5], after learning from data, it also produces a list of attribute usages to know how the individual attributes contribute to the classifier. Similarly, the approach used in[41] describes a Selective Bayesian classifier (SBC) that simply uses only those features that $C4.5$ would use in its decision tree when learning a small example of a training set, a combination of the two different natures of classifiers. Experiments conducted on ten datasets indicate that SBC performs reliably better than Naive Bayes on all domains, and SBC outperforms $C4.5$ on many datasets of which $C4.5$ outperform NB. Another illustration for using decision tree to select best features is presented in [15] where Regularized trees penalize using a variable similar to the variables selected at previous tree nodes for splitting the current node.

2.2 Challenges in using Decision Trees

Despite the fact that decision trees-based learning algorithm is one of most powerful and popular machine learning algorithms, there are also some debatable issues involving apply decision tree in solving practical problems:

- Decision tree might produce over-complex trees that do not generalize data well. They yield good performances on training data but bad results on unseen data. To overcome this, pruning process is often applied after nodes splitting stage. Control the trees depth, the minimum number of samples in each leaf or the minimum number of each split are also common ways to prevent overfitting.

- Decision trees can be unstable because a small variation added to the data set can result in a totally different tree being generated. However, we can tackle this problem for data sets some of whose attributes have pretty much variance by using an ensemble of trees.

- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.

- Decision tree learners can create biased trees if some class dominate in the data set. It is therefore recommended to balance the data prior to fitting a decision tree. There are number of methods to deal with the unbalanced class problem. *Oversampling* is a way that replicas instances with smaller quantity while *undersampling*, on the contrary, removes instances belonging to dominating classes. Another common way is to set different mis-classification for different classes to punish wrong predictions.

Chapter 3

An overview on Decision tree models

Decision tree models use a very common strategy for learning, that is *divide and conquer*. In particular, they split data into separating partitions and then fit a specific learner to each partition. The data splitting process is repeated recursively from the beginning state of data set, which makes the root node, to partitions containing a few of instances in leaf nodes. Its stopping criteria is usually when the variation or the number of instances in a node exceeds a pre-determined threshold. This top-down induction of decision trees use hill-climbing approach to search for the best way to split the current data, although it might end up in a local optima place. Classification trees are designed for response variables that take a finite number of ordered or discrete values, with the cost measured by mis-classification rate. Regression trees are for response variables that take numeric or continuous values, with prediction error typically measured by squared difference between observed values and predicted values. To make decision tree learner more stable, more robust and less sensitive to extreme values (outliers) in the data set, an ensemble of tree learners is utilized, often in a combination with sampling techniques. In this chapter, we introduce the core of decision tree algorithms for classification and regression tasks, along with their variations in a variety of optimized algorithms. Then, we mention to the concept of ensemble of trees and its power in machine learning problems.

3.1 Traditional approach to construct a decision tree

There are three main questions need to be solved when building a decision tree. They are:

- Which feature chosen to split among available features? Which measure should be employed in splitting process ?
- When the splitting nodes procedure should be stopped? Or in other words, we determine the stopping criteria of building tree process. It could be when tree depth reaches a certain number, or when the number of samples in a leaf less than a specified threshold.
- Given a leaf node in tree, which label is assigned to samples that fall into that leaf/partition.

The first published classification tree algorithm is THAID [24]. Employing a impurity measure based on observed distribution of target variable Y, THAID determines a exhaustively search all over features space X to choose a split that minimizes the sum of relative impurities at children nodes. If a variable X_i takes discrete values, a possible split might contain two subsets of those values, otherwise it could be an interval between two ordered values. The process is applied recursively on the data in each child node. Splitting stops if the relative decrease in impurity is below a prespecified threshold. Algorithm below gives the pseudocode for the basic steps.

Algorithm THAID: pseudocode for constructing decision tree by exhaustively search.

```
1. Start at root node
2. For each variable $X$ in features space, find the set $S$ that minimizes the
   sum of impurities of child nodes and then choose the split that minimizes
   that number over $X$ and $S$.
3. If the stopping criteria meets, terminate the process. Otherwise, apply
   step 2 and 3 to every child nodes in turn.
```

C4.5 [39] and CART [12] are two later classification tree algorithms that follow this approach. C4.5 uses entropy for its impurity function, whereas CART uses a generalization of the binomial variance called the Gini index. So, what is an impurity function?

3.1.1 Goodness of fit and Impurity function

Every time we examine features space for choosing a node to split the tree, we want it to be the best split, meaning that it can most likely reduce the error rate among candidate nodes. Respect to the manner of measuring *quality* of such splits, we define function $\phi(S,T)$ to evaluate goodness of fit of each split candidate S in node T. The split that maximizes the value of the goodness of fit is chosen to be the next node of tree. To define the goodness of fit function, we need to go through a new concept called impurity function. The impurity function measures the extent of purity for a region containing data points from possibly different classes. Suppose the number of classes is k. Then the impurity function is a function of $p_1, \dots, p_k$, the probabilities for any data point in the region belonging to class 1, 2,..., k. There are many ways to define the impurity function, but minimum requirements need to be satisfied. An impurity function is a function I that takes k (number of classes) parameters $p_1, \dots, p_k$, $p_i \geq 0$, $\sum p_i = 1$ and has three following properties:

- I achieves maximum only for the uniform distribution, that is all the p_j are equal.
- I achieves minimum only at the points (1, 0, ... , 0), (0, 1, 0, ... , 0), ..., (0, 0, ... , 0, 1), i.e., when the probability of being in a certain class is 1 and 0 for all the other classes.
- I is a symmetric function of $p_1, \dots, p_k$.

So assuming that there are k classes in the data set, an impurity function $I(T)$ at node T has this form

$$I(T) = I(p(1|T), p(2|T), ..., p(k|T) \tag{3.1}$$

where $p(j|T)$ is the estimated posterior probability of class j given a point in node T. Once we have impurity function for node T as $I(T)$, we can calculate the goodness of fit denoted by $\phi(S,T)$ as following:

$$\phi(S,T) = \Delta I(S,T) = I(T) - p_L I(T_L) - p_R I(T_R) \tag{3.2}$$

where $\Delta I(S,T)$ defines the difference between the impurity measure of node T and two child nodes T_R, T_L according to split S. p_L and p_R are the probabilities an instance going to left branch and right branch of T according to split S. There are several impurity functions used in tree-based model for classification:

- Entropy function: $\sum p_j log \frac{1}{p_j}$. Entropy is used as split criterion in C4.5 algorithm
- Miss-classification rate : $1 - max_j p_j$
- Gini index : $\sum_{j=1}^{N} p_j(1-p_j)$

In CART classification tree, Gini index is used as a impurity function for splitting nodes. Another splitting method is the Twoing Rule [12]. This approach does not have anything to do with the impurity function. The intuition here is that the class distributions in the two child nodes should be as different as possible and the proportion of data falling into either of the child node should be balanced. At node t, choose the split s that maximizes:

$$\frac{p_L p_R}{4}[\sum_j (p(j|t_L) - p(j|t_R))^2] \tag{3.3}$$

When we break one node to two child nodes, we want the posterior probabilities of the classes to be as different as possible. If they differ a lot, each tends to be pure. If instead the proportions of classes in the two child nodes are roughly the same as the parent node, this indicates the splitting does not make the two child nodes much purer than the parent node and hence not a successful split. To answer the question number 2 as aforementioned, an example is that the CART tree is grown until the number of data in each terminal node is no greater than a certain threshold, say 5, or even 1. For question number 3, each leaf in CART classification tree produces a label that appears the most among instances coming to that leaf. Error estimated at each leaf node is the probability of misclassifying training instances at that leaf, or in other words, $r(t) = 1 - max_j pj$. The misclassification rate at leaf node$r(t)$ is called resubsitution estimate for node t. So resubsitution estimate for the whole tree T will be calculated as follows:

$$R(T) = \sum_{t \in T_{leaf}} r(t)p(t) \tag{3.4}$$

where $p(t)$ is the number of instances at that leaf divided by the total instances in data set. In terms of difference between the three splitting criteria Entropy, Gini index and Twoing in growing trees, Breiman in [13] has revealed some interesting aspects. For example, the optimum split for the Gini criterion sends all data in the class with the largest p_j to left sub tree and all other classes to the right sub tree. Thus the best Gini splits try to produce pure nodes. But the optimal split under the entropy criterion breaks the classes up into two disjoint subsets $\gamma_1, \gamma_2 \subset \{Y_1, Y_2, \dots Y_k\}$ such as γ_1 minimizes $|\sum p_j - .5|$ among all subsets $\gamma \subset \{Y_1, Y_2, \dots Y_k\}$. Thus, optimizing the entropy criterion tends to equalize the sample set sizes in left and right children nodes. The twoing criteria also tries to equalizes the sample size [13]. In problems with a small number of classes, all criteria should produce similar results. The differences appear in data where number of classes is larger. Here, high up in the tree, Gini may produce splits that are too unbalanced. On the other hand, the above results show a disturbing facet of the entropy and twoing criterion, i.e. a lack of uniqueness [13].

3.1.2 Overfitting in Decision trees

Overfitting is a term used when a model is trained and perform much better on training data than on testing data, or in other words model doesn't generalize well on unseen instances. There are several reasons that lead to overfitting. It could be the lack of training data and thus model can not be trained to recognize patterns only existed in testing data. Another possibility comes from data quality, when some noises are added and models are too sensitive to these noises and loose their generalization ability. In contrast to overfitting, underfitting happens when the model can not represent well the complexity of training data. Underfitting causes a poor performance on both training data and testing data. Figure 3.1 illustrates cases of underfitting and overfitting regarding to the model complexity. The ideal range for selected model should lie in between.

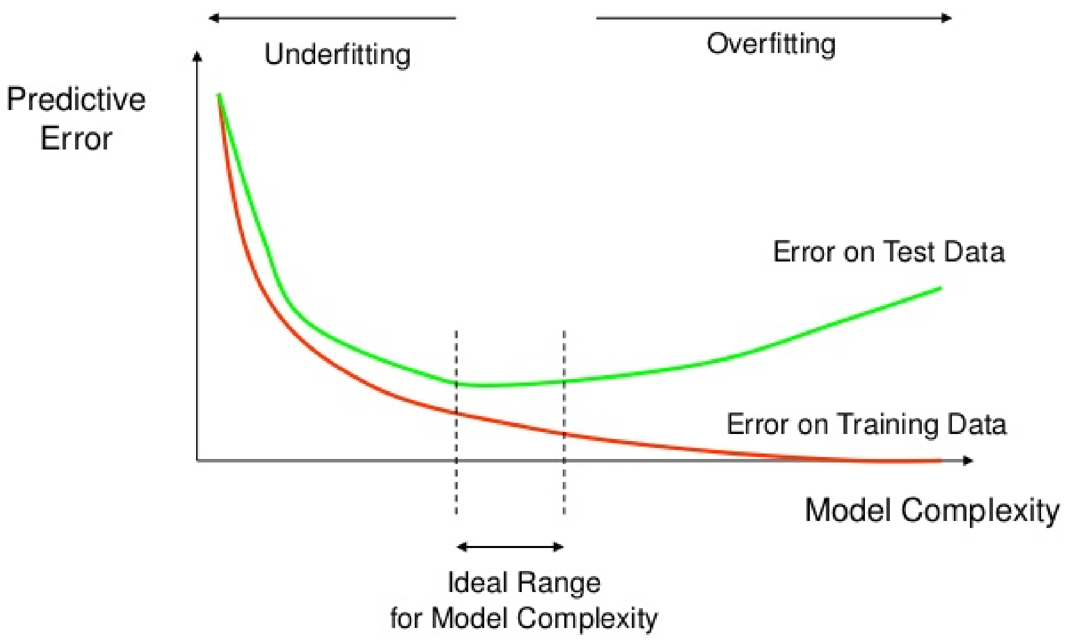


Figure 3.1: Overfitting and Underfitting regarding to model complexity

Frank (2000) has proposed an interesting perspective on overfitting problem in decision tree learners. It was well explained in [19] by the difference on sampling variance between training samples and true population of data. Frank (2000) argued that in classification tasks, where performance is measured by the number of correct predictions on future data, assigning the most populous class in the sample to future data will maximize the number of correct predictions. However, accuracy degrades if the majority class in the sample differs from the most likely class in the population. Because of sampling variance this is particularly likely if the sample is very small [19]. Frank has showed that given p_i, the prior proportion of class values in the true population, the estimated error decrease as the number of samples in leaf nodes increase.

Along with the overcomplex model that causes overfitting as shown in figure 3.1, *oversearching*, a term proposed by Quinlan and Cameron-James (1995), is also mentioned as a possibility. Employing impurity functions as Gini and entropy, tree learners always tend to search for disjunct spaces that contain *pure* training clusters, in other words clusters that only consist of training samples of the same class. And search increases the likelihood of finding apparent patterns that are caused by sampling variance and do not reflect the true underlying population proportions.

An experiment has been conducted by authors in [19] to illustrate this finding. Given a simple sample set with one numeric attribute and one response variable that takes two class A and B, there was two classifiers trained and then evaluated: one random classifier that predicts class randomly and one tree learner that has minimum training error. Figure 3.2 shows the average expected error for space of different sizes for both the randomly generated classifier and the minimum error classifier when 100 training instances are used. It could be observed that expected errors of the minimum error classifier increases and surpass that number from the random classifier when the number of samples in partitions decreases. Only for large disjunct space the discrepancy vanishes.

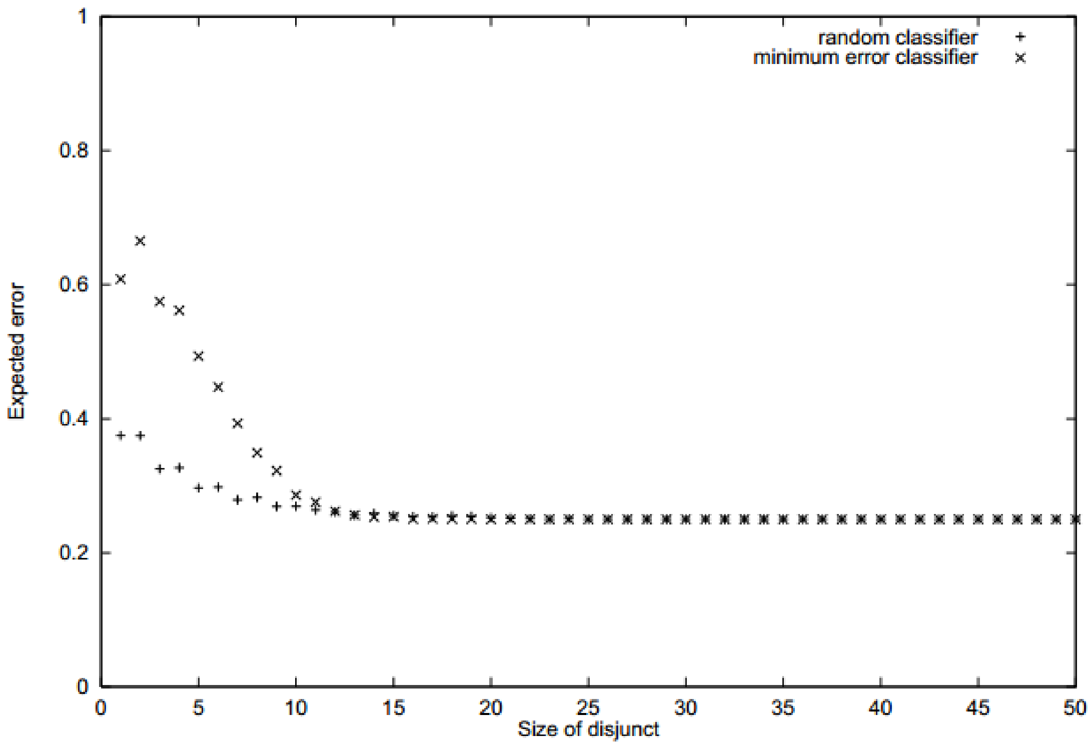


Figure 3.2: Oversearching causes overfitting for tree learners

We know that potential complex learners combined with intensive search might lead to overfitting. In decision tree learners, complexity and search are often linked because these classifiers proceed according to a general to specific paradigm: they start with the simplest possible classifier and iteratively extend it by adding more and more partitions until the training data is fully explained. To prevent overfitting, a possible way is to control the complexity of decision trees by setting thresholds for learning parameters, i.e tree depth, the maximum of samples in leaf nodes, the minimum of samples to split. Doing this way can also limit the search process and necessarily merge small partitions to bigger ones. However it is hard to decide optimal values for those parameters without using a grid search that takes a lot of time for big data sets. Trees pruning provides a way to eliminate unreliable parts more systematically and more reasonably.

3.1.3 Pruning decision trees

Pruning simplifies decision tree model by merging adjacent space of training samples. It discards parts of the learner that describe the unnecessary variation of training sample rather than true mapping regulations between features and response variable. This makes the model more comprehensible to the users, and potentially more accurate on testing instances that have not been used in training.

There are two paradigms for pruning in decision trees, pre-pruning and post-pruning. Pre-pruning procedure does not really 'prune' the tree, it stops the growth of a branch if additional structure is expected not improve the accuracy. On the other hand, post-pruning waits until the tree is fully grown and then it cuts off branches that do not improve predictive performance. An important step in pruning decision trees is to define a criterion that is used to determine the optimal pruned tree among candidates. Following we discuss some published pruning approaches, highlight their target function that needs to be optimized as well as their properties.

Pessimistic error pruning

Pessimistic error pruning was first introduced by Wild and Weber (1995) [47]. It is based on the estimated error from training data. More particularly, pessimistic error pruning adds a constant to the training error of a sub tree by assuming that each leaf automatically classifies a certain fraction of an instance incorrectly . This fraction is taken to be 1/2 divided by the total number of instances covered by the leaf, and is called a *continuity correction* in statistics [47]. The normal distribution is used to closely approximate the binomial distribution in the small sample case. A branch is pruned if the adjusted error estimate at that branch node is smaller or equal to the adjusted error of the branch. However, an assumption needs to be hold that the adjusted error is binomially distributed.

Minimum-error pruning

Minimum-error pruning was first published as works by Niblett and Bratko [34]. It shares a common strategy from pessimistic-error pruning that it uses class counts derived from training data. In its initial version by Mingers (1989), there was an adjustment to these counts to more closely reflex a leaf 's generalization performance. That is a straightforward instantiation of the Laplace correction, which simply adds one to the number of instances of each class when the error rate is computed. Minimum-error pruning proceeds in a bottom-up manner, replacing a subtree by a leaf if the estimated error of subtree is no smaller than for the leaf. In order to derive an estimate of the error rate for a subtree, an average of the error estimates is computed for its branches, weighted according to the number of instances that reach each of them

Cost-complexity pruning

Cost-complexity pruning was used in CART (Classification And Regression Tree) system by Breiman et. al. (1984) [12]. CART uses a criterion named error-complexity measure to estimate grade of nodes for pruning. As presented in the name, this measure is composed of the resubsitution estimate (or misclasifying rate) and the complexity of subtrees before and after pruning. Let α be a real number called the complexity parameter, the cost complexity measure $R_\alpha(T)$ is defined as:

$$R_\alpha(T) = R(T) + \alpha|T_{leaf}| \tag{3.5}$$

where $|T_{leaf}|$ is the number of leaves of tree T.
The more leaf nodes that the tree contains the higher complexity of the tree is because we have more flexibility in partitioning the space into smaller pieces, and therefore more possibilities for fitting the training data. There's also the issue of how much importance to put on the size of the tree. The complexity parameter α adjusts that. At the end, the cost complexity measure comes as a penalized version of the resubstitution error rate. This is the function to be minimized when pruning the tree. In general, given a pre-selected α , we find the subtree $T(\alpha)$ that minimizes $R_\alpha(T)$.

Error-based pruning

The key idea of this pruning method is to estimate error of subtrees on unseen cases so that the subtree is pruned if not improve the testing performance. The error-based pruning has been the most widely used approach. However, it depends on whether the task is classification or regression and the way authors calculate estimated errors to have variant methods. For example, M5 model tree [45] for regression estimates testing error as following: training error at each leaf node is multiplied with a heuristic number $(n+v)/(n-v)$, where n is the number of instances and v is the number of parameters at that node, to produce the estimated error at that node.

On the other hand, C5.0 for classification uses statistical learning to estimate errors for unseen cases at subtrees [5]. In this method, Normal distribution is applied to estimate the upper limit of error on unseen cases, or true error rate, with a specific confidence , based on the sample error rate (which is the training error). In statistic, a succession of independent events that either *succeed* or *fail* is called a *Bernoulli process*. Result of a test of an estimator on some instance, say X, could be a *success* or *failure*, so we can consider X as a Bernoulli distributed random variable.

Assume we have N trials consisting of N Bernoulli distributed random variables X. Out of N trials, if there are F failures the F is a random variable following Binomial distribution. The question is, how can we estimate the *true error rate* F/N given a confidence level and a sample error rate. We call the true error rate p, so the mean and variance o a single Bernoulli trial with failure rate p and $p(1-p)$, respectively. Out of N trials taken from a Bernoulli process, the expected error rate $f = \frac{F}{N}$ is a random variable with mean p and variance $\frac{p(1-p)}{N}$. For large N, the distribution of this random variable follows Normal distribution. We know that for a Normal distributed random variable with mean 0 and variance unit, we can find a value of z given a confidence level c:

$$Pr[x > z] = c \tag{3.6}$$

After converting f to satisfy having mean 0 and unit variance, we choose a c value (default $c = 0.25$) and find confidence limit z such that:

$$Pr[\frac{f - p}{\sqrt{p(1-p)/N}} > z] = c \tag{3.7}$$

where N is the number of samples, $f = F/N$ is the observed error rate, and p is the true error rate. This leads to an upper confidence limit for p. This upper confidence limit is used to estimate the error rate at each node:

$$e = \frac{f + \frac{z^2}{N} + z\sqrt{\frac{f}{N} - \frac{f^2}{N} + \frac{z^2}{4N^2}}}{1 + \frac{z^2}{N}} \tag{3.8}$$

This is how C5.0 decision tree estimates error on unseen cases at leaves nodes. Error at an internal node is a sum of error rate at its leaves nodes multiplying with instances ration going to a leaf. Pruning decision is made when comparing these two values.

3.2 Classification trees

In a classification problem, we have a training sample with n observations on a response variable Y that take k different (or discrete) values, i.e Y_1, Y_2, $\dots Y_k$ and p predictor variables $(X_i)_{i=1}^p$. The goal is to find a model for predicting values of Y from new X values. In theory, a classification decision tree should partition X space into k discrete sets, A_1, A_2, $\dots A_k$ such that instances falling in to set A_i are labeled as Y_i.

C4.5 [39] and CART [12] are two of most widely known classification trees. The impurity function for C4.5 and CART are entropy and Gini index, respectively. Although they employ the same approach of building decision tree with THAID [24] (as in Algorithm THAID), they first grow an overly large tree and then prune it to a smaller size to minimize an estimate of the misclassification error. Different from the intensive search over the data space for pure partitions that lead to overfitting, the exhaustive search strategy to find the best split also has an undesirable property. Note that an ordered variable with m distinct values has $(m - 1)$ splits of the form $X \leq c$, and an unordered variable with m distinct unordered values has $(2^{m-1} - 1)$ splits of the form $X \in S$. So a variable with more discrete values have greater possibility to be selected and this becomes a serious problem for the decision tree induction.

Building on an idea that originated in the FACT algorithm [31], CRUISE [27], GUIDE [29] and QUEST13 [30] have tackled this problem by using a two-step approach based on statistical significance tests for splitting nodes process. First, each X variable is tested for association with target variable Y and the most significant variable is selected. Then, an exhaustive search over all possible splits of all available features is performed to choose the best one . Because every X has the same chance to be selected if each is independent of Y, this approach is effectively free of selection bias. Besides, much computation is saved as the search for best split is carried out only on the selected X variable. GUIDE and CRUISE use chi squared tests, and QUEST uses chi squared tests for unordered variables and analysis of variance (ANOVA) tests for ordered variables. Pseudocode for the GUIDE algorithm is given in Algorithm below. The CRUISE, GUIDE, and QUEST trees are pruned the same way as CART.

Algorithm GUIDE: pseudocode for constructing GUIDE decision tree [29]

```
1. Start at root node
2. If $X$ is an ordered variable (taking numeric values), convert it to an unordered variable $X'$ by grouping its values in the node into a small number of intervals.
3. Perform a chi squared test of independence of each $X'$ variable versus $Y$ on the data in the node and compute its significance probability.
4. Choose the variable $X*$ associated with the $X'$ that has the smallest significance probability.
5. Find the split set $X* \in S*$ that minimizes the sum of Gini indexes and use it to split the node into two child nodes.
6. If a stopping criterion is reached, exit. Otherwise, apply steps 2–5 to each child node.
7. Prune the tree with the CART method
```

In addition to above trees, CHAID [22] follows a different strategy. If a variable X takes numerical values, it is split into 10 intervals and one child node is assigned to each interval. Otherwise, one child node is assigned to each discrete value. CHAID uses significance tests and Bonferroni corrections to try to merge couples of child nodes iteratively. However, this method is biased toward selecting variables X with fewer distinct values.

While CART and C4.5 split only one variable at a time, CART, QUEST and CRUISE can allow splits on linear combinations of numerical variables. In terms of proceeding missing values, CART and CRUISE apply the same approach that use alternate splits on other variables when needed, C4.5 provides each sample with a missing value in a split through every branch using a probability weighting scheme, QUEST imputes the missing values by mean or median, and GUIDE treats missing values as a separating category. All models accept user-defined misclassification costs. This means the cost for mis-classifying categories of the target variable might be different. By default, all algorithms fit a constant model to each node, predicting Y to be the class with the smallest misclassification cost. CRUISE can optionally fit bivariate linear discriminant models and GUIDE can fit bivariate kernel density and nearest neighbor models in the nodes. GUIDE also can produce ensemble models using bagging and random forest techniques [28].

To make a comparison on how classification tree models perform on real dataset, an experiment was conducted in [28]. The selected dataset is new car data 1993 [1]. There are 93 cars and 25 variables. Response variable takes three values (rear, front, or fourwheel drive). Three are unordered (manuf, type, and airbag, taking 31, 6, and 3 values, respectively), two binary valued (manual and domestic), and the rest ordered. The class frequencies are rather unequal: 16 (17.2%) are rear, 67 (72.0%) are front, and 10 (10.8%) are four-wheel drive vehicles. Results are presented in figure 3.3 derived from [28] (RPART stands for CART tree).

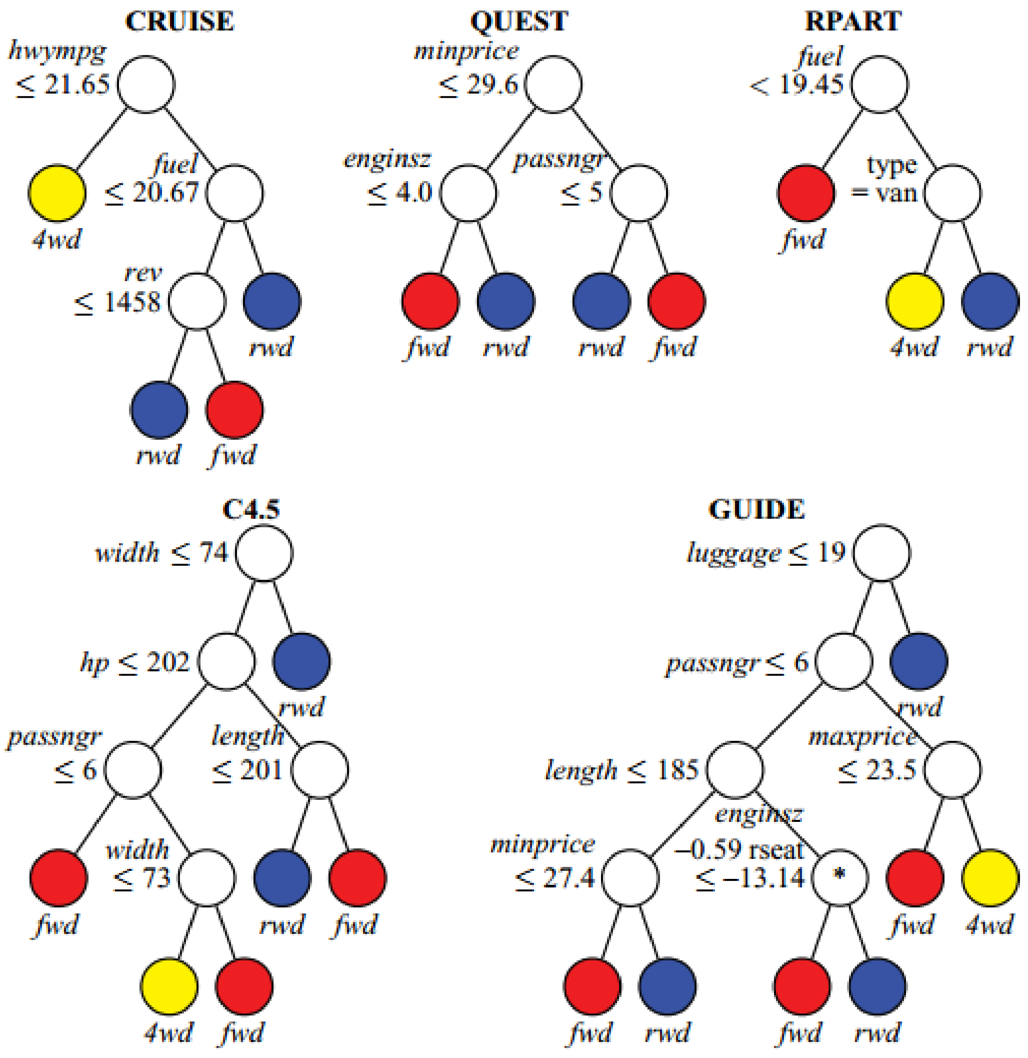


Figure 3.3: Classification trees perform on new car 1993 dataset

We can see that CRUISE, QUEST and CART produce less complex tree than C4.5 and GUIDE. Tree by QUEST looks most balanced. When GUIDE does not find a suitable variable to split a node, it looks for a linear split on a pair of variables. One such split, on *enginsz* and *rseat*, occurs at the node marked with an asterisk (*) in the GUIDE tree.

3.3 Regression trees

A regression tree is similar to a classification tree, except that the Y variable takes ordered values and a regression model is fitted to each node to give the predicted values of Y. The first regression tree algorithm published is AID [16]. The AID and CART regression tree methods follow *Algorithm THAID*, with the difference that the impurity function is standard deviations of Y variable of observations. The label value assigned to a leaf node is the mean of response values of samples that reach that node. This yields piecewise constant models. Although they are simple to interpret, the prediction accuracy of these models often lags behind that of models with more smoothness (like M5 Model tree). However, it can be computationally impracticable to extend this approach to piecewise linear models, because two linear models (one for each child node) must be fitted for every candidate split.

M5' [46] presents an adaptation of a regression tree algorithm by Quinlan, uses a more computationally efficient strategy to construct piecewise linear models. It first constructs a piecewise constant tree and then fits a linear regression model to the data in each leaf node. The regression version of GUIDE utilizes its classification techniques to solve the regression problem. At each node, it fits a regression model to the data and computes the residuals. Then a response variable $Y*$ is defined to take values 1 or 2, depending on whether the sign of the residual is positive or not. Finally, it applies *Algorithm GUIDE* to the $Y*$ variable to split the node into two. This approach has three advantages: (1) the splits are unbiased; (2) only one regression model is fitted at each node; and (3) because it is based on residuals, the method is neither limited to piecewise constant models nor to the least squares criterion.

While employing variance or standard deviation of response values is a reasonable approach for splitting nodes and most of regression decision trees use this approach, there is quite a diversity in the way they perform the pruning process. The main difference comes from the target function they would like to optimize. Two of most used functions are *Cost-complexity* as in CART [12] and *Error-based pruning*

3.3.1 Pruning in CART regression tree

Regression CART tree after be built is pruned to avoid overfitting on unseen cases. It uses a similar approach with classification CART tree, meaning that it aims to minimize the cost-complexity criterion.

$$R_\alpha(T) = R(T) + \alpha|T_{leaf}| \tag{3.9}$$

where cost function $R(T)$ is no longer misclassification rate but mean square error between observed target variable and its estimated values. This procedure results in a series of complexity parameters and corresponding sub trees:

$$\begin{gathered} T > T_2 > T_3 > ... > t \\ \alpha_1 < \alpha_2 < ... < \alpha_v \end{gathered} \tag{3.10}$$

The question remaining is that how we can choose the best pruned tree from the above set. Briemann [12] has used cross validation approach to produce the best tree. Assume we split the dataset into V subsets, L_1, L_2,..., L_v with as equal as possible number of instances. We also let the training sample set in each fold be $L^{(v)}$. First, a tree is grown on the original data set, denoted by T_{max}, and we repeat this process on each subset $L^{(v)}$, denoted by $T^{(v)}_{max}$.

For each complexity parameter α_i of T_{max}, we will let T_α and $T^{(v)}_\alpha$ be the cost-complexity minimal subtree of T_{max} and $T^{(v)}_{max}$. Notice that for any $k \geq 1$, $\alpha_k \leq \alpha < \alpha_{k+1}$, the smallest optimal subtree $T_(\alpha)^{(v)} = T_(\alpha_k)^{(v)}$, i.e., is the same as the smallest optimal subtree for α_k.

From here we can calculate the cross validation error for each complexity parameter α:

$$R(T_\alpha) = \frac{1}{N}\sum_{i=1}^{V}\sum_{x_j,y_j \in L_i}(y_j - T^{(v)}_\alpha)^2 \tag{3.11}$$

Breiman and his colleagues have described two alternatives for the final tree selection based on the obtained error estimates. Either to select the tree with lowest estimated error or to choose the smallest tree in the sequence, whose error estimate is within the interval $R(T_\alpha)_{min} + S.E(R(T_\alpha))$, where $R(T_\alpha)_{min}$ is the lowest error estimate and $S.E(R(T_\alpha))$ is the standard deviation error of this estimate. This latter method is usually known as the 1-SE rule, and it is known to favor simpler trees although possibly leading to lower predictive accuracy

3.3.2 Error-based pruning for regression trees

The Error-based pruning method tries to estimate the error at each node itself and error at subtree below on unseen test cases then compare two numbers to give the pruning decision. However, there is still not any *best* way that is widely accepted to compute estimated errors. Following are a few proposed ways in well-known regression tree models.

M5 Model tree

In M5 tree, training error at each leaf node is multiplied with a heuristic number $(n + v)/(n - v)$, where n is the number of instances and v is the number of parameters at that node, to produce the estimated error at that node.

m-estimator

This is a Bayesian method to estimate a population parameter using the following combination between our posterior and prior knowledge:

$$mEST(\theta) = \frac{n}{n+m}\zeta(\theta) + \frac{m}{n+m}\pi(\theta) \tag{3.12}$$

where, $\zeta(\theta)$ is our posterior estimation of the parameter (based on a size n sample), $\pi(\theta)$ is our prior estimate of the parameter and m is a parameter of this type of estimators. Cestnik and Bratko(1991) [14] used this method to estimate class probabilities in the context of post-pruning classification trees using the $N\&B$ pruning algorithm. Karalic and

Cestnik (1991) [26] extended this framework to the case of l east squares (LS) regression trees. These authors have used m-estimators to obtain reliable tree error estimates during the pruning phase. Obtaining the error of an LS tree involves calculating the mean squared error at each leaf node. As a result, we can obtain the m estimate of the mean and MSE in a leaf l by,

$$\begin{aligned} mEst(y_{mean}) &= \frac{1}{n_l+m}\sum_{i=1}^{n_l} y_i + \frac{m}{n_l+m}\sum_{i=1}^{n} y_i \\ mEst(MSE_{y_{mean}}) &= \frac{1}{n_l+m}\sum_{i=1}^{n_l} y_i^2 + \frac{m}{n_l+m}\sum_{i=1}^{n} y_i^2 - mEst(y_{mean})^2 \end{aligned} \tag{3.13}$$

Estimates based on Sampling Distribution Properties

For the MSE criterion, the error associated with a leaf can be seen as an estimate of the variance of the cases within it. Statistical estimation theory tells us that the sampling distribution of the variance is the χ^2 if the original variable follows a normal distribution. According to the properties of the χ^2 distribution, a $100(1-\alpha\%)$ confidence interval for the true population variance based on a sample of size n is given by:

$$(\frac{(n-1)s_Y^2}{\chi^2_{\frac{\alpha}{2},n-1}}, \frac{(n-1)s_Y^2}{\chi^2_{1-\frac{\alpha}{2},n-1}}) \tag{3.14}$$

where, s_Y^2 is the sample variance (obtained in a particular tree leaf); and $\chi^2_{\alpha,n}$ is the tabulated value of the χ^2 distribution for a given confidence level α and n degrees of freedom. This formulation is based on an assumption of normality of the distribution of the variable Y. In most real-world domains we cannot guarantee a priori that this assumption holds.
The χ^2 distribution is not symmetric, meaning that the middle point of the interval defined by the above equation does not correspond to the sample point estimate of the variance. In effect, the middle point of this interval is larger than the point estimate. The difference between these two values decreases as the number of degrees of freedom grows, because it is known that the chi^2 distribution approximates the normal distribution when the number of degrees of freedom is sufficiently large. This means that as the sample size (which corresponds to the number of degrees of freedom) grows, the middle point of the interval given in Equation above will tend to approach the point estimate obtained with the sample. This is exactly the kind of bias most pruning methods rely on. They "punish" estimates obtained in the leaves of large trees (with few data points) when compared to estimates at higher levels of the trees. Being so, authors in [43] propose using the middle point of the interval in equation above as a more reliable estimate of the variance of any node, which leads to the following estimator of the MSE in a node t:

$$ChiEst(MSE(t)) = MSE(t) \times \frac{n_t-1}{2} \times (\frac{1}{\chi^2_{\frac{\alpha}{2},n-1}}, \frac{1}{\chi^2_{1-\frac{\alpha}{2},n-1}}) \tag{3.15}$$

3.4 Ensemble of trees as a powerful method

Generally, the purpose of ensemble methods is to combine the predictions of several base estimators constructed with a particular learning algorithm in order to improve generalized ability and robustness over a single estimator. Ensemble of trees is a case in the family of ensemble methods where base classifiers are decision trees. Follow the distinguished branches of ensemble methods, we can view ensemble of trees in two typical divisions.

1. In averaging methods, the driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced. Examples for this kind of ensemble of trees are Random Forest, Bagging.

2. By contrast, in boosting methods, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble. Two examples are AdaBoost (with base estimator being Decision Tree) and Gradient Tree Boosting.

Following we illustrate why ensemble of trees model can perform better than individual estimator itself in terms of the accuracy and avoiding overfitting. We consider Bagging representing the first branch and AdaBoost representing second branch.

3.4.1 Bagging

Bagging methods build several instances of a black-box estimator on random subsets of the original training set and then aggregate their individual predictions to form a final prediction. More over, bagging introduces randomization into forming procedure and then making an ensemble out of it. Randomness is embedded in two ways: randomly choose instances and randomly choose subset of features. This way is proved to be able to reduce overfitting of a base estimator (decision tree) [23]. We know that strategy *divide and conquer* of decision tree might cause over-searching and the difference between distribution of a data subset between training/testing data leads to overfitting. Subsampling technique in bagging decreases the possibility of encountering such bad sectors of data thus reduces overfitting. Ensemble of trees is also expected to reduce the variance.

3.4.2 AdaBoost

The main idea of AdaBoost is to fit a sequence of weak learners (such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction [20]. For each training iteration, the sample weights are individually adjusted and the learning algorithm is reapplied to the data with new set of weights. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence [20]. That is the reason why AdaBoost yields a much higher performance than individual decision trees.

It is the fact that ensemble of tree methods methods, like Random Forest, has become more and more popular due to its efficiency in practical Machine Learning problems.

Chapter 4

M5 Model Trees

4.1 M5 model tree

M5 model tree is a decision tree learner for regression task, meaning that it is used to predict values of numerical response variable Y. M5 tree was invented by J.R.Quinlan (1992). While M5 tree employs the same approach with CART tree [12] in choosing mean squared error as impurity function, it does not assign a constant to the leaf node but instead it fit a multivariate linear regression model; the model tree is thus analogous to piecewise linear functions. According to [40], M5 model tree can learn efficiently and can handle tasks with very high dimensionality - up to hundreds of attributes. This ability sets M5 apart from regression tree learners at the time (like MARS), whose computational cost grows very quickly when the number of features increases. Also, the advantage of M5 over CART is that model trees are generally much smaller than regression trees and have proven more accurate in the tasks investigated [40].

4.1.1 Construct M5 tree

Follow the recursively node-splitting strategy of decision trees as described in Chapter 3, M5 tree partitions the data into a collection of set T and the set T is either associated with a leaf, or some test is chosen that splits T into subsets corresponding to the test outcomes and the same process is applied recursively to the subsets. This process often produces over-elaborate structures, namely overfitting, that must be pruned.

Information gain in M5 tree is measured by the reduce in standard deviation before and after the test. First step is to compute the standard deviation of the response values of cases in T. Unless T contains very few cases or their values vary only slightly, T is split on the outcomes of a test. Let T_i denote subset of cases corresponding to ith outcome of a specific test. If we treat deviation $sd(T_i)$ of target values of cases in T_i as a measure of error, the expected reduction in error can be written as follows

$$\Delta error = sd(T) - \sum_i \frac{|T_i|}{|T|} sd(T_i) \tag{4.1}$$

Then M5 tree will choose one that maximizes this expected error reduction. To compare, CART chooses a test to give the greatest expected reduction in either variance or absolute deviation [12].

4.1.2 Pruning M5 tree

Pruning is proceeded in a manner from leaves to root node. At each internal node, M5 tree compares the estimated error of that node and estimated error of subtree below. Then the subtree is pruned if it does not improve performance of the tree.

Error-based estimate

M5 model tree uses error-based method for pruning. The key factor of this method is to estimate the error/accuracy of the model on unseen cases. M5 tree computes this number by first averaging the absolute difference between response values of observations and predicted values. This will generally underestimate the error on unseen cases, so M5 multiplies it by $(n + v)/(n - v)$, where n is the number of training cases and v is the number of parameters in the model [40]. The effect is to increase the estimated error of models with many parameters constructed from small number of cases.

The estimated error of a subtree is calculated as sum of estimated error of the left and right tree below that node multiplying with the proportion of samples that goes down to left and right tree.

Linear models

A multivariate linear model is fit into each node of the tree using standard regression technique. However M5 tree does not use all features in the dataset, instead it is restricted to features that are referenced by tests or linear models in the subtrees below this node. As M5 will compare the accuracy of a linear model with the accuracy of a subtree, this ensures a level playing field in which the two types of models use the same information [40].

After the linear model is built, M5 tree simplifies it by greedily eliminate coefficients one by one. This way might generally result in the increase in the averaged residual, however it also reduces the multiplicative factors above, so the estimated error can decrease.

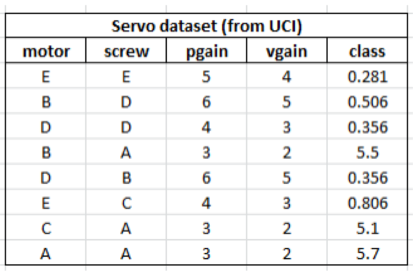
4.1.3 Smoothing

Pregibon observes that the prediction accuracy of tree-based models can be improved by a smoothing process. When the value of a case is predicted by a model tree, the value given by the model at the appropriate leaf is adjusted to reflect the predicted values at nodes along the path from the root to that leaf. The form and motivation of smoothing process in M5 is inspired by Pregibon. It is described as follows:

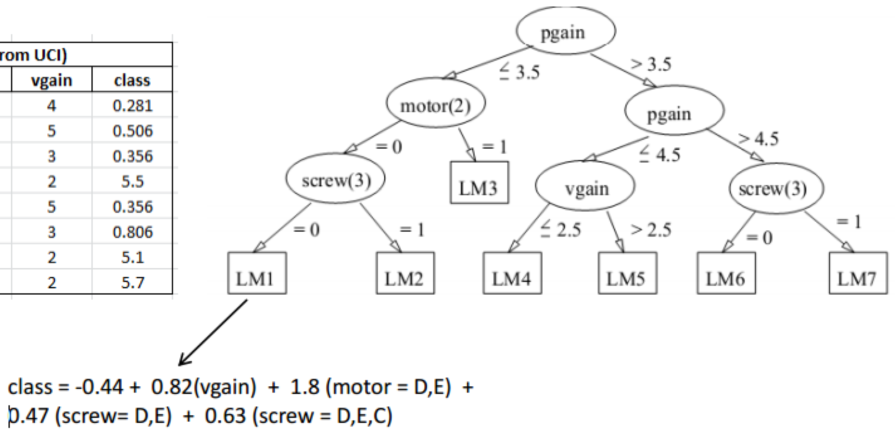
- The predicted value at the leaf is the value computed by the model at that leaf
- If the case follows branch S_i of subtree S,let n_i be the number of training cases at S_i, PV(S_i)the predicted value at S_i ,and M(S)the value given by the model at S. The predicted value backed up to S is

$$PV(S) = \frac{n_i.PV(S_i) + k.M(S)}{n_i + k} \tag{4.2}$$

To illustrate how M5 model tree works on a real problem, let take a look on Servo dataset (UCI repository). There are 4 features consisting of two categorical variables *motor*, *screw* and two numerical variables *pgain* and *vgain*. The class variable takes real values whose range is from 0.13 to 7.1. A part of this dataset and the complete M5 tree for this dataset are presented in Figure 4.1.



Servo dataset (from UCI)				
motor	screw	pgain	vgain	class
E	E	5	4	0.281
B	D	6	5	0.506
D	D	4	3	0.356
B	A	3	2	5.5
D	B	6	5	0.356
E	C	4	3	0.806
C	A	3	2	5.1
A	A	3	2	5.7



class = -0.44 + 0.82(vgain) + 1.8 (motor = D,E) + 0.47 (screw= D,E) + 0.63 (screw = D,E,C)

Figure 4.1: M5 model tree performs on Servo dataset

4.2 M5' model tree

M5' model tree was first proposed by Yong Wang and Ian H. Witten (1997). M5' is a new implementation of model-tree inducer based on M5 tree by Quinlan [40]. Some improvements include modifications to reduce tree size, thus the tree becomes more comprehensible. M5' also introduces a way to deal with categorical variables and missing values. According to [46], M5' performed somewhat better than the original algorithm on the standard datasets for which results have been published.

To clarify the form of linear model that was not clear in M5 tree, M5' uses $k + 1$ parameters where k is the number of attributes in data set and 1 represents constant term. Second, during the initial splitting procedure, M5' does not split a node if it represents three examples or less or the standard deviation of the class values of the examples at the node is less than 5% of the standard deviation of the class values of the entire dataset. Third, M5' decides to leave attributes, that are dropped from a model when their effect is so small that it actually increases the estimated error, in higher-level models if they are useful.

Fourth, all enumerated attributes are transformed into binary variables before constructing a model tree. The average class values corresponding to each possible value in the enumeration is calculated from the training examples, and the values in the enumeration are sorted according to these averages. Then, if the enumerated attribute has k possible values, it is replaced by $k - 1$ synthetic binary attributes, the ith being 0 if the value is one of the first i in the ordering and 1 otherwise [46]. To tackle the problem that enumerated attributes having a large number of different values are automatically favored, M5' multiplies the SDR value by a factor that is unity for a binary split and decays exponentially as the number of values increases.

In terms of processing missing values, M5' modify the formula of SDR to

$$SDR = \frac{m}{|T|} \times \beta(i) \times [std(T) - \sum_{j \in L,R} \frac{|T_j|}{|T|} \times sd(T_j)] \tag{4.3}$$

where m is the number of instances without missing values for that attribute, and T is the set of examples that reach this node. $\beta(i)$ is the correction factor mentioned above, computed for the original attribute to which this synthetic attribute corresponds. T_L and T_R are the sets that result from splitting on this attribute. The pseudocode of M5' model tree algorithm is given below

```
M5'(examples)
{
  SD = sd(examples)
  for each k-valued enumerated attribute
       convert into k-1 synthetic binary attributes
  root = new_node
  root.examples = examples
  split(root)
  prune(root)
}
split(node)
{
 if sizeof(node.examples) < 4 or sd (node.examples) < 0.05 * SD
  node.type = LEAF
 else
  node.type = INTERIOR
  for each continuous and binary attribute
   for all possible split positions
    calculate the attribute's SDR
  node.attribute = attribute with max SDR
  split(node.left)
  split(node.right)
}
prune(node)
{
 if node = INTERIOR then
  prune(node.left_child)
  prune(node.right_child)
  node.model = linear_regression(node)
  if subtree_error(node) > error(node) then
   node.type = LEAF
}
```

4.3 M5'Rules

M5'Rules was invented by Holmes et. al. [21]. Technically, it generates a rules set from M5' model tree following this process: a tree learner (M5' tree) is applied to the full training data set and a pruned tree is learned. Next, the best leaf (according to some heuristic) is made into a rule and the tree is discarded. All instances covered by the rule are removed from the dataset. The process is applied recursively to the remaining instances and terminates when all instances are covered by one or more rules. The rules are generated based on an unsmoothed linear models. To illustrate this process of generating rules from a model tree, figure 4.2 presents a model tree built for *bolts* dataset (1999) and then a set of rules are produced from this tree.

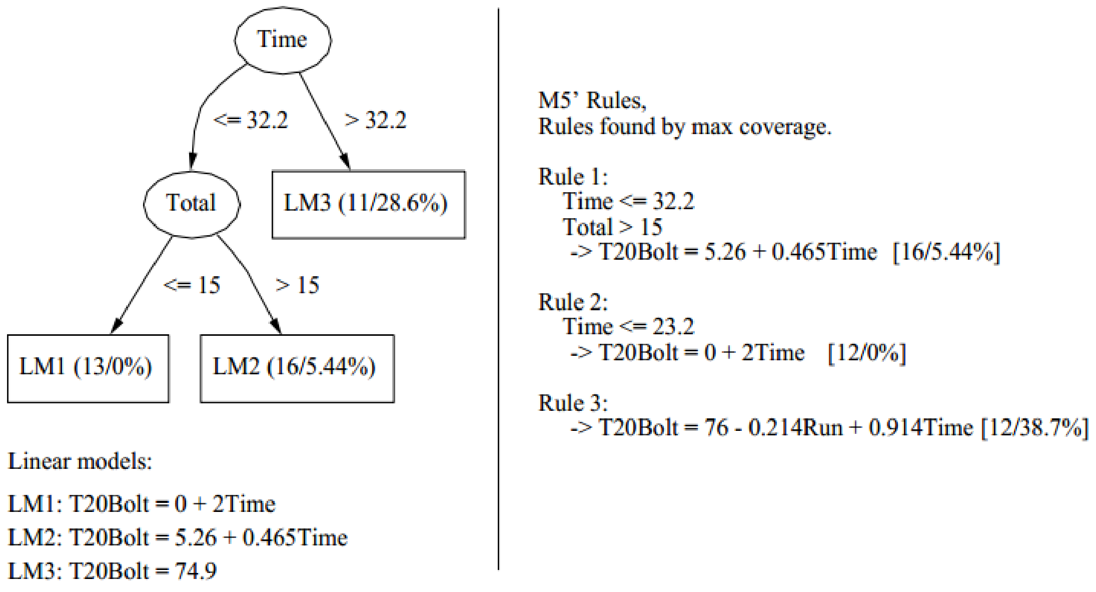


Figure 4.2: M5'Rules for *Bolt* dataset

So we know that at each stage, M5'Rules choose a *best* node and convert it to a rule. The question is which criteria to compare 'quality' of nodes and then pick the most informative one. In [21], they have proposed 4 heuristics. The first and most obvious one is to choose the leaf which covers the most examples. Figure 4.2 actually uses this heuristic to extract set of rules. The second one calculates the percent root mean squared error as shown below:

$$\%RMS = \frac{\sqrt{\sum_{i=1}^{N_r}(Y_i - y_i)^2/N_r}}{\sqrt{\sum_{i=1}^{N_r}(Y_i - \bar{Y})^2/N}} \tag{4.4}$$

where Y_i is the actual class value for example i, y_i is the class value predicted by the linear model at a leaf, N_r is the number of examples covered by leaf, $\bar{Y}$ is the mean of the class values, and N is the total number of examples. In this case, small values of % RMS (less than 1) indicate that the model at a leaf is doing better than simply predicting the mean of the class values. One potential problem with percent root mean squared error is that it may favor accuracy at the expense of coverage.

The third and fourth show two measures designed to trade of accuracy against coverage. The third, simply normalizes the mean absolute error at a leaf using the number of examples it covers; the fourth, multiplies the correlation between the predicted and actual class values for instances at a leaf by the number of instances that reach the leaf.

$$MAE/Cover = \frac{\sum_{i=1}^{N_r}|Y_i - y_i|}{2N_r} \tag{4.5}$$

$$CC.Cover = \frac{\sum_{i=1}^{N_r} Y_i y_i}{N_r \sigma_Y \sigma_y} N_r \tag{4.6}$$

Authors in [21] also conducted a experiment to test the performance of all 4 listed heuristic. They came to several conclusions that M5'Rules using coverage heuristic is better that M5' in terms of accuracy, M5'Rules never produces larger rule sets and produces smaller sets on tested datasets. When compared to the commercial system Cubist, M5'Rules outperforms it on size and is comparable on accuracy. When based on the number of leaves it is more than three times more likely to produce significantly fewer rules.

4.4 M5 model tree for classification

Although M5 model tree was originally invented for regression tasks, it also proves efficient when derived to handle classification problems. This approach was proposed in [18] by E. Frank et. al. In this approach, a data set whose target variable consists of k discrete values is split into k data sets each of which has the same set of independent variables S and one dependent variable Y_i representing a value of Y. More particularly, target value of a sample in ith data set is 1 if that sample belongs to Y_i class and 0 otherwise. For example in figure 4.3, the well-known Iris dataset is transformed into 3 sub-dataset for the *Setosa*, *Virginica* and *Versicolor* varieties of Iris. Then k different model trees are built for the k datasets. For a specific instance, the output of one of these model trees constitutes an approximation to the probability that this instance belongs to the associated class. Since the output values of the model trees are only approximations, they do not necessarily sum to one.

In the testing stage, an instance of unknown class is processed by each of the model trees, the result being an approximation to the probability that it belongs to that class. The class whose model tree gives the highest value is chosen as the predicted class.

4.5 Multivariate Linear Regression in M5 model tree

One of most important properties of a M5 model tree is that it uses a multivariate linear regression at each leaf to predict numerical values. Besides, linear regression model also is constructed at internal nodes in pruning process. Therefore, performance of a M5 model tree counts mainly on how good those linear models are. In this section of our work, we would like to introduce what is linear regression model, well-known approaches used to solve this problem and our selected approach in implementing M5 model tree. Most of knowledge presented in below sections is from [38]

4.5.1 Linear regression model

To find a multivariate linear regression model for a data set, which holds a M instances containing values of a group of N attributes and values of one class, we have to solve a set of linear algebraic equations like this:

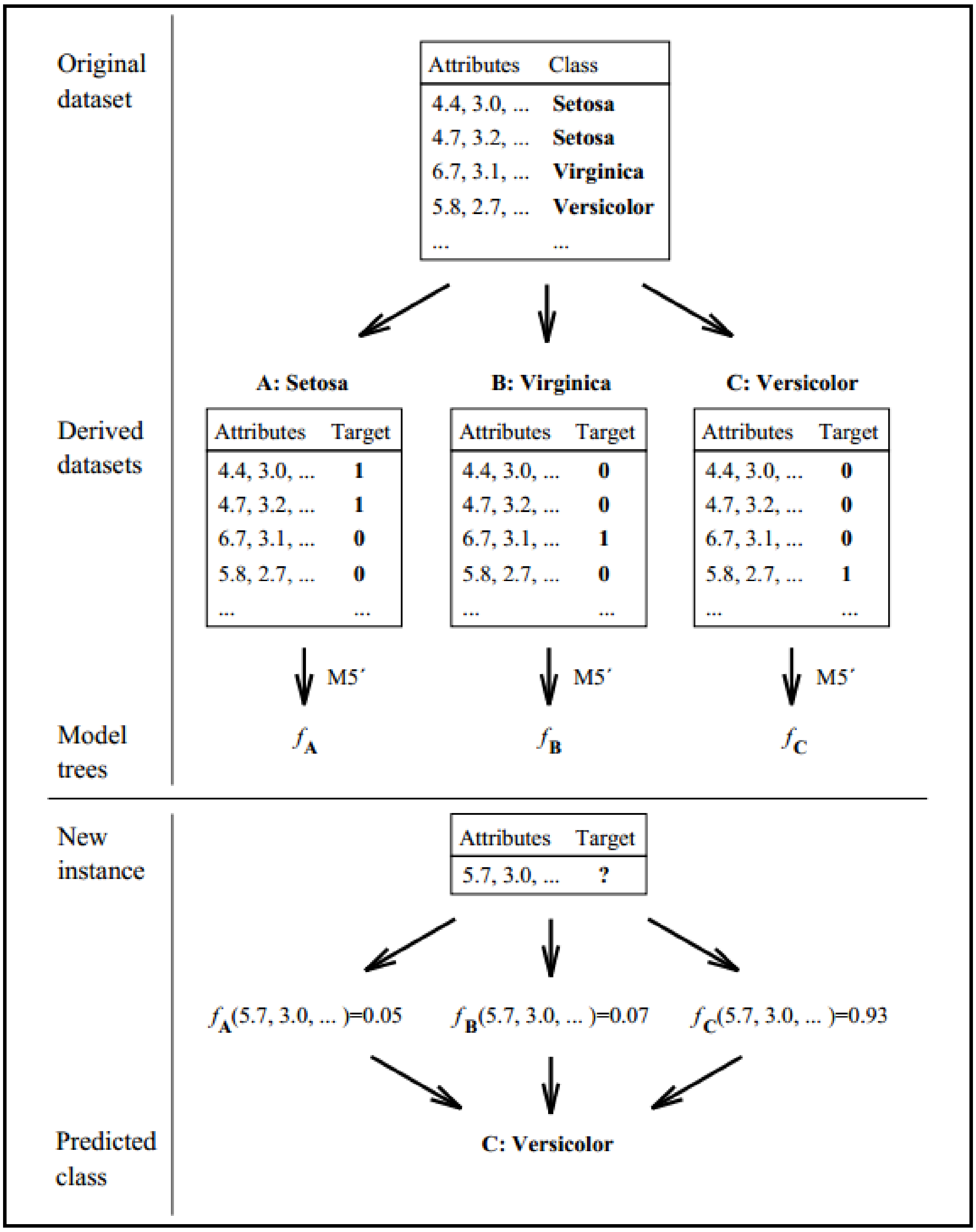


Figure 4.3: How to use M5 tree in classification with Iris dataset

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + ... + a_{1N}x_N &= b_1 \\
a_{21}x_1 + a_{22}x_2 + ... + a_{2N}x_N &= b_2 \\
a_{31}x_1 + a_{32}x_2 + ... + a_{3N}x_N &= b_3 \\
&.... \\
a_{M1}x_1 + a_{M2}x_2 + ... + a_{MN}x_N &= b_M
\end{aligned}$$

Here the N unknowns, x_j, j = 1,2,...,N are related by M equations. The coefficients a_{ij} with i = 1,2,...,M and j = 1,2,...,M are known numbers collected from values of attributes of data set. The *right-hand side quantities* b_i with i = 1,2,3...,M are from class values.
If N = M then there are as many equations as unknown and there is a good chance of solving for a unique solution set of x_j's. However, if one or more of M equations is a linear combination of the others, a condition called *row degeneracy*, or if all equations contain certain variables only in exactly the same linear combination, called *column degeneracy*, it can be fail to find a unique solution. A set of equations that is degenerate is called *singular* [38]. This is an important concept is Linear Algebraic Equations and we have separating methods to deal with those cases.
Set of linear equations as above can be written in matrix form as

$$\mathbf{A}\,.\,\mathbf{x} = \mathbf{b} \tag{4.7}$$

Here **A** is the matrix of coefficients and **b** is the right-hand side written as a column vector.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \cdots & & & \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{bmatrix} \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_M \end{bmatrix}$$

To solve the set of linear equations, one of simplest and thus most common strategies is to use Linear least-square estimation. $M \times N$ set of equations as presented in (1) can be written as the $N \times N$ set of equations

$$(A^T.A).x = (A^T.b) \tag{4.8}$$

The Linear least-square estimation method minimizes the sum of squared residuals, and leads to a closed-form expression for the estimated value of the unknown x. The problem now becomes solving a $N \times N$ set of equations as in (2) and this offers an advantage rather than $N \neq M$ because there is a good chance of solving for a unique solution of unknowns. There are a lot of methods for solving this kind of problem. Here we present the three of most popular ones: *Gauss-Jordan Elimination*, *Gausian Elimination with Backsubstitution* and *LU Decompostion*

4.5.2 Gauss-Jordan Elimination

This method can be used to find a matrix inversion and also to solve sets of linear equations. For a better illustration of the method, we will write out equations for only the case of four equations and four unknowns and one right-hand side vector of four elements.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

The routine of Gauss-Jordan elimination is processed as follows [38]. The first row is divided by the element a_{11} and this is a trivial linear combination of the first row with any other row - zero coefficient for the other row. Then each other row is subtracted an amount of first row multiplying their first elements. As a result, the first column of A now agrees with the identity matrix. We move to the second column and divide the second row by a_{22}, then subtract the first, third and fourth row from an amount of second row multiplying their second element, so as to make their entries in the second column zero. And so on for the remaining rows and A is transformed to a identity matrix. As we do the operations to A, we also do the corresponding operations to the b's.
In this routine, the element that we divide by is called the *pivot.* We can interchange rows *(partial pivoting)* or rows and columns *(full pivoting)* in order to put a particularly desirable element in the diagonal position from which the pivot is about to be selected. There is an obvious problem when one of elements on diagonal, or pivot, is zero and we can not divide the row by it. This is a case of solving a singular matrix.
In terms of finding matrix inversion, we just have to apply the routine above to solve 4 equations

$$A.x_1 = b_1 \quad A.x_2 = b_2 \quad A.x_3 = b_3 \quad A.x_4 = b_4 \tag{4.9}$$

where b_i, i = 1, 2, 3 ,4 are columns of identity matrix. The resulting matrix is built by x_i, i = 1, 2, 3 ,4 as columns.

4.5.3 Gaussian Elimination with Backsubstitution

Use similar routine as Gauss-Jordan elimination, the only difference is that matrix A is reduced not all the way to identity matrix but only half way. At each stage, we subtract away rows only below the current pivot element. When a_{22} is the pivot element, for example, we divide the second row by its value (as before), but now use the pivot row to zero only a_{32} and a_{42}, not a_{12} [38]. The equation after reduction routine will be:

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix}$$

Notice that here original b matrix is also modified according to operations made to turn matrix A to A' in above equation. The solving of x now becomes quite easy following these formula

$$x_4 = b'_4/a44' \tag{4.10}$$

$$x_3 = \frac{1}{a'_{33}}[b'_3 - x_4 a'_{34}] \tag{4.11}$$

then we proceed x before x_3 with

$$x_i = \frac{1}{a'_{ii}}[b'_i - \sum_{j=i+1}^{N} a'_{ij} x_j] \tag{4.12}$$

The procedure defined in (6) is called *backsubstitution.* The combination between Gaussian elimination and backsubstitution yields a solution for solving set of linear equations. The advantage of this method over Gauss-Jordan elimination is simply that this one is faster in raw operation count [38].

4.5.4 LU Decomposition

Suppose matrix **A** can be decomposed into two matrices, a lower triangular matrix **L** and a upper triangular matrix **U**

$$\mathbf{L}.\mathbf{U} = \mathbf{A} \tag{4.13}$$

In case of 4 x 4 matrix, equation above looks like this:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & a_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

To solve the linear of equations problem, (7) can be written as

$$A.x = (L.U).x = L.(U.x) = b \tag{4.14}$$

We first solve the equation

$$L.y = b \tag{4.15}$$

and then solve

$$U.x = y \tag{4.16}$$

To solve equations in (9) and (10), we can use procedure (5) described in section of Gaussian elimination and backsubstitution. The remaining problem is to solve the decomposition and find **L** and **U**. Equation in (7) is broke into sum of terms of α_{ij} and β_{ij}

$$i < j : \quad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{4.17}$$

$$i = j : \quad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{4.18}$$

$$i > j : \quad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ij}\beta_{jj} = a_{ij} \tag{4.19}$$

Because there are $N^2 + N$ unknown but only N^2 equations in equations from (11), (12) and (13), we can possibly assume that

$$\alpha_{ii} = 1 \quad i = 1, 2, ..., N \tag{4.20}$$

Then we use *Crout's algorithm* to solves the set of $N^2 + N$ equations for all α 's and β 's by just arranging the equations in a certain order as follows.

- Set $\alpha_{ii} = 1$, i = 1, 2, ..., N
- For each j = 1, 2, 3, ... ,N do two procedures: First, for i = 1, 2, ..., j, use (11), (12) and (14) to solve for β_{ij}

  $$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \tag{4.21}$$

  Second, for i = j + 1, j+2, ..., N use (13) to solve for α_{ij}

  $$\alpha_{ij} = \frac{1}{\beta_{ij}}(a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}) \tag{4.22}$$

4.5.5 Solve a singular matrix

There is a well-known method to take over of singular matrix, it is *Singular Value Decomposition*: a $M \times N$ matrix A whose number of rows M is greater than or equal to its number of columns N, can be written as the product of an $M \times N$ column orthogonal matrix **U**, an $N \times N$ matrix diagonal matrix product of an $M \times N$ orthogonal matrix **V**.

$$[A] = [U] . \begin{bmatrix} w_1 & & & \\ & w_2 & & \\ & & \cdots & \\ & & \cdots & \\ & & & w_N \end{bmatrix} . [V^T]$$

then

$$A^{-1} = V.[diag(1/w_j)].U^T \tag{4.23}$$

where we replace $1/w_j$ by 0 if w_j is nearly zero. So we can solve x like this:

$$x = V.[diag(1/w_j)].(U^T.B) \tag{4.24}$$

However, in practice, SVD is not an optimal method because it takes a lot of resources. In our implementation, we use LU decomposition method in a combination with a modified matrix A to solve set of linear equations. Each time A is proved as a singular matrix, a pre-determined constant, namely *ridge* is added to each element in the diagonal of matrix A to form a new matrix which is no longer singular. The *ridge* is small enough to ensure the change made to A is really trivial.

```
int success = 0;
double ridge = 10^(-8);
int i ,j;
while (success == 0){
 info = LAPACKE_dgesv( LAPACK_ROW_MAJOR, n, nrhs, &A[0],
  lda, &ipiv[0], &B[0], ldb );
 /* Check for the exact singularity */
 if( info > 0 ) {
  for (i = 0; i < n; i++){
    A [i*n + i] = A[i*n + i] + ridge;
  }
  ridge = ridge * 10.0;
 }
 else
  success = 1;
}
```

Chapter 5

Introduction of ADATE system

In the first part of this chapter, we give a brief presentation to concepts of Evolutionary algorithm in Machine Learning and Automatic programming. Then in the second part, we introduce the ADATE system as an implementation and founded based on these above concepts. Because the limited space of this dissertation, we only focus on main properties and analytic power of ADATE. For a further and deeper understanding of ADATE system, we highly recommend readers to refer [35], [36] and [37]

5.1 Evolutionary algorithm and Automatic programming

5.1.1 Evolutionary algorithm

Artificial evolution is a well-known approach in machine learning. This approach attempts to create a new generation of hypothesis based on a collection of current hypothesis, called current population. Some of elements in the current population are added directly into the new set of hypothesis, whereas remaining ones are replaced by offsprings using transformation rules. Generally, this is a learning method that is inspired from the biology evolution. A typical and popular representative for artificial evolution in machine learning is Genetic Algorithms (GAs).

Rather than search from general-to-specific hypotheses, or from simple-to-complex, GAs generate successor hypotheses by repeatedly mutating and recombining parts of the best currently known hypotheses. GAs identifies the best hypothesis using a function named 'fitness function'. The process forms a generate-and-test beam-search of hypotheses, in which variants of the best current hypotheses are most likely to be considered next. A short description of GAs is as follows. At each iteration, a new set of hypothesis is created by selecting probabilistically in the current population hypothesis which are most fit and by adding new hypothesis. New hypotheses are created by applying a crossover operator to pairs of most fit hypotheses and by creating single point mutations in the resulting generation of hypotheses. This process is iterated until sufficiently fit hypotheses are discovered.

To achieve a deeper insight into GAs, some theoretical points need to be discussed clearly. And we present point by point in details as following.

Hypotheses representing

In general, hypotheses in GAs are represented by bit string in which each element is either 0 or 1. The reason for this could be that GAs algorithm will be easier to apply offspring generation rules such as cross over or mutation. The idea of representing hypotheses by bit string was introduced for the first time by Holland (1986), Grefenstette (1988) and DeJong et al. (1993) [33].

To illustrate for this, consider an example of turning the statement *"If weather is cool and I feel happy, I will play football"* into a bit string. Assume that attribute *weather* has three values *rain*, *cool* and *sunny*. So the first condition *weather is cool* would be encoded by string 010 where second bit stands for *cool*. Similarly, assume *feeling* attribute has only two values, either*happy* or *sad*, the second condition should be encoded such as 10. And the resulting part of this statement is represented by string 01 that is composed by *No* and *Yes* values. So the whole statement is encoded as a bit string that is 0101001.

Fitness function and parents selection The output of fitness function is used to rank potential hypotheses and also to probabilistically select them for inclusion into the new generation. This is also why values of fitness function should be positive.

There will be a part of parents taken into the next generation. These parents are selected probabilistically, with the probability of a string being selected being proportional to its fitness.

Offspring generation GAs uses genetic operators to generate offspring from parents. These operators are close to idealized version of genetic operation found in biological evolution [33]. Two popular genetic operators in GAs are *cross-over* and *mutation*. In *cross-over*, there are two offsprings produced from two parent strings by copying selected bits from each parent. It is an additional string named *cross-over mask* to decide which parent to the i(th) bit should belong to. There are three kinds of *cross-over* described in Figure 5.1 which was also used in [33].

- Single-point crossover : As illustrated in Figure 5.1, imagine there is a point separating the crossover mask. This makes the first offspring taking the first five bit from the first parent and remaining bits from the other one. For the second offspring, the process repeats with switched parents' role.
- Two-point crossover : Similar to the single-point crossover, the crossover mask is separated by 2 points. The i(th) bit of the offspring will decide get the i(th) bit from either first or second parent based on value of i(th) bit from crossover mask string.
- Uniform crossover : In this case, the crossover mask bits string is created randomly.

For the *mutation* operator, the offspring produced by changing the value of a random bit from a single parent, as the illustration in Figure 5.1.

5.1.2 Automatic programming

Automatic programming is defined as synthesis of programs that solve problems described in a specification. There will be no more programming. The end users only need to have knowledge about the application domain. They write a specification file to describe what they want. The automatic programming system, which only knows how to program, will produce an efficient program that solve the problem in specification file [42]

Also in [42], there are three key features in an automatic programming system: 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.

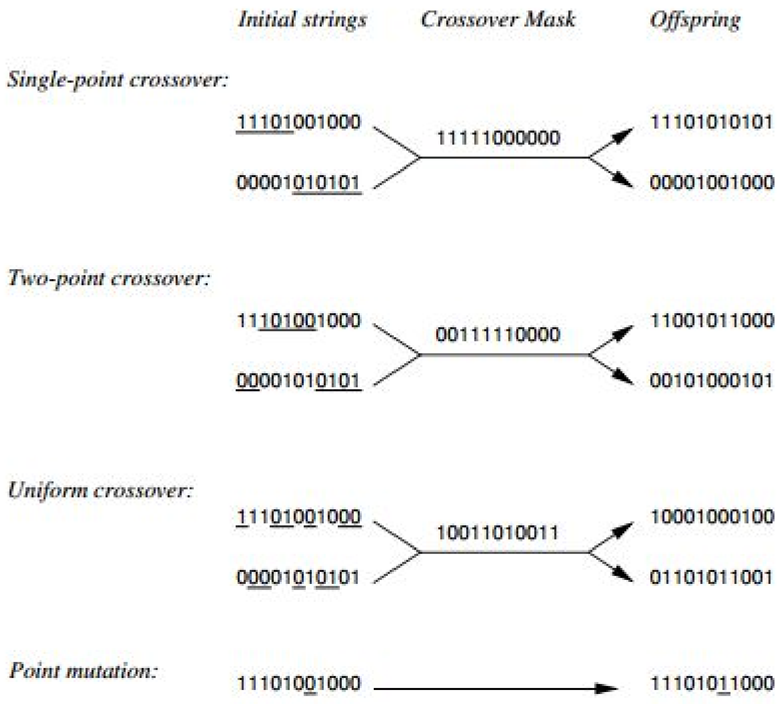


Figure 5.1: Genetic operators: Cross-over and mutation

According to Biermann in [11], *"Computer programming is the process of constructing executable code from fragmentary information. ... When computer programming is done by a machine, the process is called automatic programming. AI researchers are interested in studying automatic programming for two reasons: First, it would be highly useful to have a powerful automatic programming systems that could receive casual and imprecise specifications for a desired target program and then correctly generate that program; second, automatic programming is widely believed to be a necessary component of any intelligent system and is therefore a topic for fundamental research in its own right."*

5.2 Introduction to ADATE

5.2.1 What is ADATE system?

ADATE (Automatic Design of Algorithms Through Evolution) is an automatic programming system developed by Professor Roland Olsson at Ostfold University College. ADATE is used to automatically generate solution programs that can solve specified problems through a series of inductive inference of algorithms.

The specification file that is the input to ADATE system must contain data samples for the problem, that is, inputs and corresponding outputs of the specified problem. Furthermore, it should specify an output evaluation function. This function is similar to the fitness function aforementioned in Genetics Algorithm, that is used to decide if a generated program is acceptable and put into population or not. The way ADATE works is to search in the hypotheses space for programs that fit criteria revealed in specification file. A series of heuristics is utilized to increase the efficiency of the search.

The collection of programs by ADATE is called the kingdom where each program is considered as a individual. At the beginning of the search, a kingdom has only one individual that is empty and raises only an exception. Then new individuals are generated by applying compound transformation rules. A compound transformation rule is composed of basic forms of transformations. There are six basic forms of transformations, called atomic transformations as described below. Definition of rules are cited from [37]

- R (Replacement): Replacement changes part of the individual with new expressions.
- REQ (Replacement without making the individuals evaluation value worse): Does the same as Replacement but now the new individual is guaranteed to have an equal or better evaluation value.
- ABSTR (Abstraction): This transformation takes an expression in the individual and puts that expression in a function in a let . . . in block and replaces the expression with a call to that function.
- CASE-DIST (Case distribution): This transformation takes a case expression inside a function call and moves the function call into each of the case code blocks. It can also change the scope of function definitions.
- EMB (Embedding): This transformation changes the argument type of functions.
- Crossover: This operator views a number of REQ transformations as alleles and recombines them using a genetic algorithm operating on fixed-length genomes.

To be able to placed into the kingdom, a generated individual should perform better than programs that already exist in the kingdom. ADATE uses a evaluation function, with the same role as fitness function in GAs, to calculate value of individuals. ADATE compares a newly generated individual with the individuals in the family it belongs to and possibly replaces them if they have worse evaluation values.

By default, the evaluation values are calculated based on the number of correct and wrong training examples, the number of memory, and time limit overflows. However, ADATE system also allows users to define GRADE structure that makes evaluation values taking into account user defined grade. An illustration for GRADE and evaluation function will be presented in the next section.

It is the fact that the evolution in ADATE system is inspired by basic biological principles of evolution, like for instance mutation and crossover. Artificial evolution process in ADATE and natural selection share common following properties:

- Generated off-springs should be superior to their parents in certain criteria. For instance, natural selection requires next generations adapting the living environment more than their parents can. Similarly, in ADATE, successive programs should produces a better performance on the evaluation function than their ancestors.

- There are similarities between transformation rules of ADATE and the way to create variations in natural selection. For example, replacement rule and cross-over rule in ADATE can be viewed as biological mutation and gene permutation in natural selection.

- These two processes might last forever without stop as long as successive generations are better than the current ones.

However, artificial evolution used in ADATE also highlights some differences from natural evolution. The most significant distinction is that ADATE uses heuristic to limit the search space and time, thus increase the efficiency of the search. In contrast, the major source variation used in natural evolution, mutation, is produced randomly and does not follow any principles. As a matter of fact, the latter process takes much longer than the former one. In return, due to a larger search space without using heuristic, there is a huge set of instances of off-springs generated from natural evolution and so this increases the chance to produce best individuals.

To conclude this section, we present here a brief overview of the evolution and induction flow during the run of ADATE system:

1. Initiate the kingdom with the single program given as the start program in the specification file (either an empty program or some program that is to be optimized). In addition to the actual programs, the system also maintains an integer value C_P for each program P , called the cost limit of the program. For new programs this value is set to the initial value 1000.

2. Select the program P with the lowest C_P value from the kingdom.

3. Apply C_P compound program transformations to the selected program, yielding C_P new programs.

4. Try to insert each of the created programs into the kingdom in accordance with the size-evaluation ordering described above.

5. Double the value of C_P, and repeat from step 2.

The above loop is repeated until the user terminates the system. The ADATE system has no built-in termination criteria and it is up to the user to monitor the evolving programs and halt the system whenever he considers the evolved results good enough.

5.2.2 ADATE for classification

In classification task, a classifier has to conduct predictive analysis and produce a discrete value of response variable. To see how ADATE perform on a classification problem, we take *Iris*, a well-known dataset on UCI repository as example. This data set has 150 instances each of which contains five explanatory variables that describe properties of Irish plants like petal length, petal width, sepal length and sepal width. The response variable is a categorical whose values are *Setosa*, *Virginica* and *Versicolor* varieties of Iris.

Use ADATE to run this data set, the best synthesized program gives an accuracy of *100%* on training set and *82.7%* on the validation set, the run time was 10020 seconds. As a evolutionary system, ADATE is powerful in giving a number of solutions that are different in degree of logic order during its search through the hyper-parameter space. Three programs below are typical examples of ADATE's results for Iris dataset.

```
fun f1( X0real, X1real, X2real, X3real ) =
  case
    rconstLess( X3real, rconst( 1, 0.25125, 0.172751946272 ) )  of
    false => irisClassvirginica
  | true =>
     case realLess( X1real, rconst( 1, 0.26165, 0.51494362372 ) ) of
      false => irisClasssetosa
     | true =>
        case => realLess(X2real, rconst(1, 0.57394, 0.4534343 ) ) of
        false => irisClassversicolor
        | true => irisClassvirginica
}
```

```
fun f2( X0real, X1real, X2real, X3real ) =
  case realLess( X1real, X2real ) of
    false => irisClasssetosa
  | true =>
  case
    realLess(
      realSubtract( X2real, X3real ),
      realMultiply( X3real, X0real )
      )  of
    false => irisClassversicolor
  | true => irisClassvirginica
```

It is obvious to see that the program number 1 looks like a decision tree where each split is made by a comparison between a variable and a constant. Simple as that, we can say

```
fun f2( X0real, X1real, X2real, X3real ) =
  case realLess( X1real, X2real ) of
    false => irisClasssetosa
  | true =>
  case
    realLess(
      realSubtract( X2real, X3real ),
      realMultiply( X3real, X0real )
      ) of
    false => irisClassversicolor
  | true => irisClassvirginica
```

```
fun f3( X0real, X1real, X2real, X3real ) =
  case realLess( X1real, X2real ) of
    false => irisClasssetosa
  | true =>
  case
    realLess(
      X2real,
      realDivide( X3real, sigmoid( sigmoid( X0real ) ) )
      ) of
    false => irisClassversicolor
  | true => irisClassvirginica
```

ADATE has generated a first-order program to learn Iris dataset. Then, program number 2 becomes a bit more complicated when there are comparisons between two variables $X1real$ and $X2real$, or two expressions $X2real - X3real$ and $X3real \times X0real$. The program number 2 uses higher-order logic and thus surpass decision tree models in terms of complexity. Program 2 also shows that ADATE is capable of inferring linear combinations of variables as new features. This ability of ADATE is actually very useful in discovering higher-order features in problems that have a low level of granularity or where features are highly correlated. This kind of problem is pretty popular in practical Machine Learning use where the current set of features can not explain the complexity of data, or we can call it *underfitting* problem. In such situations the solution is normally to exploit relations between features in dataset by applying models that can capture relative information between features or manually creating higher-orders features, e.g we can create features X^2, X^3 for linear regression models.

Not only can ADATE discover new features by generating automatically expressions of current features, but also can it invent functions as presented in program 3. Here $sigmoid(sigmoid(X0real))$ is a function that takes a real number as the parameter. To invent new functions, ADATE applies the approach of evolutionary algorithm that mutation or applies cross-over operator on original functions, and usually along with abstraction transformation to guide the search process. In an example illustrated in [37], we have this function as the pre-search function, meaning before transformed:

```
fun max3( X, Y, Z ) = if X < Y then Y else X
```

After applying abstraction transformation, we have a new function:

```
fun max3( X, Y, Z ) =
 let
  fun g( V1, V2 ) = if V1 < V2 then V2 else V1
 in
  g( X, Y )
 end
```

ADATE has substituted V1 for X and and V2 for Y. This example shows that abstractions can be substitutive which means that a number of equal sub-expressions are replaced with one and the same parameter. These equal sub-expressions can be arbitrarily big and need not consist of a single variable as above.

Similar to decision tree learners, ADATE use *Divide and Conquer* strategy to construct models. It can split the input space into many sub-spaces, and build different feature extractors in different sub-spaces. This is a very powerful ability, which may explain why ADATE can easily synthesize classifier that fit very well to many different problems. However, this also makes ADATE's synthesized programs easy to be overfitted. Generally, the ADATE actually builds an ensemble of many different models in different input's sub-spaces. This divide and conquer strategy not only brings the fitting power to the ADATE system, but also introduces ovefitting problem. Splitting the input space too much will increase the chance of discovering accidental relationships (between explanatory variables and target variable as well as among explanatory variables) of ADATE system, especially when we have low-granularity training data (i.e. not enough data). Of course, the Occam's razor employed in ADATE searching strategy, and a good validation set could be very helpful in this case. Nevertheless, empirical results showed that in general, using ADATE to build a classifier for a high dimensionality continuous input domain is more likely to create an overfitted solution, compared to other use cases of ADATE. You will see in the next section that, on regression problem, ADATE usually does not divide the input space in its solutions, which make it harder to be overfitted.

5.2.3 ADATE for regression

Using its function-inventing ability, ADATE has proved efficient in finding solutions for regression problems. To understand how ADATE works on regression task, we present here two examples.

The first is a simple example where we would like to predict the velocity of a falling ball when it hits the ground, given the height H as its starting position. The correct answer for this problem is $\sqrt{19.6H}$. The solution that ADATE can come up with is shown below:

Although this ADATE's solution is a bit more complex than the original answer, which is $\sqrt{19.6H}$, Dang in [25] has explained that it is still simpler than a neural network's solution. In his experiment, he designed a neural network with 1 input, 10 hidden nodes and 1 output and then got a program containing 42 operations. He concluded that ADATE generalizes better than neural network for this SPEED problem.

In the second example, we determines to construct a model for learning Wine Quality dataset from UCI. This dataset has 11 numeric features, namely *Fixedacidity*, *Volatileacidity*, *Citricacid*, *Residualsugar*, *Chlorides*, *Freesulfurdioxide*, *Totalsulfurdioxide*, *Density*, *Ph*, *Sulphates*, *Alcohol*, and one numeric target variable *quality* whose its value ranges

```
fun f H =
 realAdd (
  realMultiply (
   H,
   tanh ( tanh ( sqrt ( tanh ( tanh ( tanh ( tanh ( tanh ( H ) ) ) ) ) ) ) ) 6 ) ,
 realAdd (
  tanh ( H ) ,
  sqrt (3)
 )
)
```

from 0.0 to 10.0. A typical program synthetized by ADATE is as follows

Compare to programs generated for classification, we notice that ADATE does not use divide and conquer strategy for regression. Or in other words, ADATE does not partition data space into sub-space and then construct particular models for them, as the way tree learners like M5 model or CART does. This prevents ADATE from overfitting due to the difference in distribution between testing data and training data in sub-spaces.

5.2.4 ADATE to handle imbalanced dataset

In this section, another powerful aspect of ADATE system is mentioned when it is utilized to tackle one of most popular issues in classification Machine Learning task, imbalanced class. Normally, a practical real-world data set is composed predominately of instances that approximately equally belong to different classes. The data set becomes imbalanced when the amount of instances of some class shows a big deviation from other classes. In the domain of skewed data set issue, a major part of cases is related to data sets that only contain two classes: the negative class or majority class and positive class which is considered as minority class. This binary class problem often appears in fields of medical diagnosis, gene profiling and credit card fraud detection.

The data set we have chosen to introduce this problem,named Amazon Employee Access, was from one of Kaggle competitions [3]. The data consists of real historical data collected in 2010 and 2011. Description of the data is described as in table 5.1. The model built should produce an output which is either 0 or 1 for attribute *ACTION* to determine whether or not access should be granted to some employee. For this data set, the number for majority class instances is 30872 while this number for minority class is 1897. This results in the a class ratio negative class : positive class that is 17 : 1.

Because the size of Amazon Employee Access data set is really big(over 32000 instances), ADATE system has to take a few weeks to run and then produce non-trivial results on a single Linux machine. Therefore, we have used a scaled version of Amazon data set with only 2000 instances. The class ratio in this scaled data set is still 16 : 1, nearly the same with the ratio in the original data set. By doing this way, we still follow our purpose that is to observe how well the ADATE system can deal with an imbalanced data set.

ADATE separates the data set into two parts of training and validation set with 1000 instances for each. There are two specification files for two experiments. The first one is generated automatically using default settings for GRADE structure and output evaluation function. In the second file, we have attempted to apply misclassifying costs by defining our own GRADE structure as well as evaluation function.

```
fun f
      (
        Fixedacidity ,
        Volatileacidity ,
        Citricacid ,
        Residualsugar ,
        Chlorides ,
        Freesulfurdioxide ,
        Totalsulfurdioxide ,
        Density ,
        Ph,
        Sulphates ,
        Alcohol
        ) =
  realMultiply(
    Alcohol ,
    realAdd(
      Volatileacidity ,
      sigmoid(
        realSubtract(
          realAdd(
            realAdd(
              let
                fun g66316 V66317 =
                  realAdd( realAdd( V66317, V66317 ), V66317 )
              in
                g66316(
                  realSubtract( Volatileacidity , Residualsugar )
                  )
              end,
              realAdd(
                realAdd(
                  realSubtract( Density, Citricacid ),
                  realSubtract( Density, Residualsugar )
                  ),
                realSubtract( Volatileacidity , Chlorides )
                )
              ),
            Volatileacidity
            ),
          Sulphates
          )
        )
      )
    )
```

Attribute name	**Description**
ACTION	ACTION is 1 if the resource was approved, 0 if the resource was not
RESOUCE	An ID for each resource
MGR_ID	The EMPLOYEE ID of the manager of the current EMPLOYEE ID record
ROLE_ROLLUP_1	Company role grouping category id 1
ROLE_ROLLUP_2	Company role grouping category id 2
ROLE_DEPTNAME	Company role department description
ROLE_TITLE	Company role business title description
ROLE_FAMILY_DESC	Company role family extended description
ROLE_FAMILY	Company role family description
ROLE_CODE	Company role code; this code is unique to each role

Table 5.1: Amazon Employee Access description

For skewed class problems, the accuracy ratio or error rate is not a appropriate measure because the result usually dominate high frequency class. So we attempted to include another measure, which is F-measure, in evaluation function. Performance of programs generated will be evaluated by both *numCorrect*, which represent accuracy ratio, and *F-measure*. We define GRADE structure as followings:

Our defined GRADE contains three real numbers. First number represents F-measure value. Two remaining number are the TP (True Positive) rate and TN (True Negative) rate, respectively. While F-measure is used in evaluating programs, TP and TN rate are printed for observation. Also in GRADE structure, we define callback functions such as **op+** (an operator that adds two grades), **comparisons** (compare two grades), **pack** (convert grade to string) and **unpack** (convert back from string to grade). For a brief description of output evaluation function, we update values of F-measure, TP and TN in grade after each iteration in which an instance is run. F-measure in this function holds a role as the role of fitness function in genetic algorithm. Run this *spec* file in about several hours, the best program generated in kingdom is presented in Code 5.1

On training data set, this program can classified correctly 99% instances. The positive rate is **87.5%**, negative rate is 100% and F-measure is **0.934**. Obviously the performance on training set is improved a lot in the comparison with the default *spec* file, which is biased on negative instances and misclasifying most of positive instances.

```
structure Grade : GRADE =
struct

type grade = real * real * real
val zero = ( 0.0, 0.0, 0.0 )
val op+ = fn( (point1,TP1, TN1) : real*real*real, (point2,TP2,TN2)
 :real*real*real) => ( point1 + point2, TP1 + TP2,TN1 + TN2)
val comparisons = [ fn( (X1,Y1,T1), (X2,Y2,T2) ) =>
  case Real.compare( X1, X2 ) of
    EQUAL => Real.compare( Y1, Y2 )
  | C => C
  ]

val pack = fn( X, Y,Z ) => "(" ^ Real.toString X ^ "," ^
  Real.toString Y ^ "," ^ Real.toString Z ^ ")"

val toString = pack

val unpack = fn S => case String.explode S of #"(" :: Xs =>
  case dropwhile( fn X => X <> #",", Xs ) of _ :: Ys =>
  case dropwhile( fn X => X <> #",", Ys ) of _ :: Zs =>

  case Real.fromString( String.implode Xs ) of SOME X =>
  case Real.fromString( String.implode Ys ) of SOME Y =>
  case Real.fromString( String.implode Zs ) of SOME Z => ( X, Y, Z)

val toRealOpt = NONE

val post_process = fn X => X

end
```

Code 5.1: Evaluation function of Amazon Access Employee problem

```
fun f
      (
        X0real,
        X1real,
        X2real,
        X3real,
        X4real,
        X5real,
        X6real,
        X7real,
        X8real
        ) =
  case realLess( X1real, X0real ) of
    false => ACTION1
  | true =>
  case realLess( X6real, X2real ) of
    false => ACTION1
  | true =>
  case
    case realLess( realAdd( X4real, X3real ), X1real ) of
      false => realLess( sigmoid( X7real ), X2real )
    | true =>
    case realLess( X3real, X7real ) of
      false => false
    | true =>
    case realLess( X0real, X3real ) of
      false => (
        case realLess( X6real, X1real ) of
          false => (
            case realLess( X1real, X3real ) of
              false => realLess( X7real, X6real )
            | true => realLess( X0real, X8real )
            )
        | true =>
        case realLess( X1real, X3real ) of
          false => (
            case realLess( X5real, X6real ) of
              false => realLess( X2real, X7real )
            | true => realLess( X7real, X0real )
            )
        | true => realLess( X0real, X8real )
        )
    | true => realLess( X6real, X8real )  of
    false => ACTION1
  | true => ACTION0
```

Chapter 6

ADATE experiments: Design and Implementation

In this chapter, we describe ADATE experiments in an investigation of possibilities to improve M5 Model tree. While results of experiments will be collected and analyzed in the next chapter, in this section we present how we choose the target for improvements, how we design experiments in the respect to our research questions, then the way we implement related libraries and specification files.

6.1 ADATE experiments design

6.1.1 Selecting targets

Before coming up with any approach in experiments design, we need to make an important decision: which part of the M5 model tree should be put into ADATE system for improvements? There are roughly three separating stages in constructing a M5 tree, that is, splitting nodes, tree pruning and smoothing process. M5 tree uses standard deviation as a metric to split nodes, then constructs linear models at internal nodes to estimate test errors in pruning stage and finally remove discontinuity effect between leaf nodes in smoothing stage. Although generally ADATE does not have any strict restrictions on the input program, we think it would be reasonable for improved programs to still remain the idea and nature of M5 model tree. So we was looking for parts that are less involved in the main idea and not clearly explained in M5 algorithm. Our selected targets are:

- *In Pruning*: To estimate the error on unseen cases of a node (actually the linear model in that node) or a subtree, M5 algorithm multiplies the training error with a function of number of instances in that node and number of involved parameters, as:

  $$Err(test) = \frac{n+v}{n-v} \times Err(train) \tag{6.1}$$

  However in known documents for M5 [40] and M5' [46] trees, there is no proof for the goodness of this approximation. Actually finding a method to estimate the testing error based on training error have been always a crucial task in *Error-based pruning*. This motivated us in using Evolutionary algorithm to search for a effective and reasonable Error-estimate function, with the above formula as started program.

- *In Smoothing*: Smoothing is used to compensate for the sharp discontinuities that will inevitably occur between adjacent linear models at the leaves of the pruned tree, particularly for some models constructed from a small number of training instances. The calculation in smoothing procedure described by Quinlan (1992) is:

  $$p' = \frac{np + kq}{n + k} \tag{6.2}$$

  where p' is the prediction passed up to the next higher node, p is the prediction passed to this node from below, q is the value predicted by the model at this node, n is the number of training instances that reach the node below, and k is the constant (default value 15). In [40] and [46], smoothing process were proved substantially increase the performance of tree. However, similar to error estimation case, there has been still not any proof that supports the smoothing function above. Thus it is possible to try finding a *better* approximation of 6.5 by Evolutionary algorithm.

6.1.2 Datasets

In ADATE experiments we use 30 datasets mainly from UCI repository [10]. They have the number of instances ranging from a few thousands to a few of ten thousands. They contain varying numbers of numeric and nominal attributes. None of them have missing value. An overview of selected datasets is introduced in Table 6.1. For each dataset to be experimented, we used K-Fold cross validation technique that is to separate the set into K different subsets, then train on $K - 1$ subsets and test on the remaining one. The process is repeated K times (folds) to get the averaged result.

6.1.3 ADATE experiments design

A. M5 smoothing improvement by ADATE

In this experiment, our target is to investigate how and to what extent ADATE can improve the performance of M5 tree learner after smoothing process. More particularly, we use ADATE in a search for a so-called *better* program than 6.5 and apply to smoothing. There are three experiments in total. The first experiment is only involved in one dataset (California House) to see how ADATE performs on a specified problem. For the second and the third experiment, we would like to make generated programs more generalized and also avoid overfitting, thus we fit all datasets in Table 6.1 to train Evolutionary model in ADATE.

Smoothing experiment 1: We choose to improve M5 tree on California House dataset because the performance of M5 on this dataset is quite poor compared to other learners, as shown in Table 6.2. So this offers a chance to possibly modify the M5 algorithm and increase its performance on a specified dataset.

In this experiment, we divide the dataset into 10 folds. For each fold, there is a M5 tree built for 9/10 data and its performance is estimated on the remaining 1/10 data. Then these 10 trees are fit into ADATE as inputs. However, ADATE is only trained on the first 5 trees, the synthesized programs are validated on the latter 5 trees.

Smoothing experiment 2: With only one dataset used to train ADATE, synthesized programs might be good for that data itself but not effective for other cases, or in other words, programs do not generalize well. This is actually the case we encounter in smoothing experiment 1. To prevent this from happening, we attempt to bring more data into ADATE system, say all 30 datasets. In this experiment, each dataset is split into training set and test set with ratio 66% : 33%, the training set is for building M5 tree and test set is used to measure its performance. Among 30 trees, 15 of them are brought into ADATE to train the evolutionary system whereas the remaining 15 are used to validate results. One might wonder why we need such many trees for validating, roughly the same number of trees for training. The answer is to overcome overfitting problem. Evolutionary algorithm normally exhaustively searchs over the hyper-parameter space looking for most-fitting programs to the training data and it usually leads to overfitting. Dealing with overfitting always keeps a high priority in ADATE.

Smoothing experiment 3: During the smoothing experiment 2, we have known that using only one tree each dataset to train ADATE programs is overfitting-prone when these programs are tested on other folds. Thus we decided to increase the number of M5 trees each dataset in this experiment. More particularly, we build 5 different trees from 5 different folds of each dataset (so we have 150 trees in total), then 100 trees are used to train ADATE programs and the remaining 50 trees are for validating. Results and related discussion will be brought in the next chapter.

B. M5 pruning improvement by ADATE

Apply the similar setting as smoothing experiment 3, the only difference is the target where we aim for improving the performance of pruning procedure in M5 tree. Here the input program would be equation 6.3.

$$Err(test) = \frac{n+v}{n-v} \times Err(train) \tag{6.3}$$

6.2 ADATE experiment implementation

This section briefly introduces the way we implement experiments described above. There are two main modules. First we need implement a M5 model tree library that can take data as input, produce a M5 tree and predict values for new cases. Then we prepare specification files representing experiments conducted in ADATE. A full description of specification file and how to use it is available in [2]. Generally, a specification file should contain requirements for the functions to be synthesized to the system. Some of important requirements include starting program, training and validating data, an evaluation function to grade programs. All modules in this section are written in MLTON [8], an optimization of functional programming Standard ML.

6.2.1 Linear model library

In this section we present our implementation for constructing the linear model in each internal node of M5 tree. An overview of approaches to solve a linear regression problem has been mentioned in Chapter 4. Here we choose the ordinary least square (OLS), the simplest and thus most common estimator. The OLS method minimizes the sum of squared residuals, and leads to a closed-form expression for the estimated value of the unknown set of coefficient:

$$x = (A^T.A)^{-1}.(A^T.b) \tag{6.4}$$

The expression 6.4 also means we have to solve a matrix equation as described in 6.5.

$$(A^T A).x = (A^T.b) \tag{6.5}$$

For an optimization of learning speed, we adopted library OpenBLAS [9] to implement matrix solving operation in 6.5. OpenBLAS is an optimized BLAS library [4] which are routines that provide standard building blocks for performing basic vector and matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software. The interface using OpenBLAS is written in C language.

To take over of singular matrix problem, we use a modified matrix A. Each time A is proved as a singular matrix, a pre-determined constant, namely *ridge* is added to each element in the diagonal of matrix A to form a new matrix which is no longer singular. The *ridge* is small enough to ensure the change made to A is really trivial. Complete code for *solve* matrix function is presented in Code 6.1

Code 6.1: Solve matrix function

```
void solve(int N, int NRHS, double* A, double* B) {
    int n = N, nrhs = NRHS, lda = N, ldb = NRHS, info;
    int ipiv[N];

        int success = 0;
        double ridge = 10^(-8);
        int i ,j;
        while (success == 0){
        info = LAPACKE_dgesv( LAPACK_ROW_MAJOR,n, nrhs, &A[0], lda,
                              &ipiv[0], &B[0], ldb );
        if( info > 0 ) {
                        for (i = 0; i < n; i++){
                                A [i*n + i] = A[i*n + i] + ridge;
                        }
                        ridge = ridge * 10.0;
                }
                else
                        success = 1;
        }
}
```

Besides matrix solving operation, we also implemented other vector and matrix operations with OpenBLAS library, such as matrix-matrix multiplying, matrix-vector multiplying and vector-vector multiplying. OpenBLAS helps saving a lot of running time for our models. To be able to use OpenBLAS functions that are written in C, we need foreign function interface of MLTON that allows accessing and manipulating C data from SML. In Code 6.2, we declare MLTON functions that make callbacks to C functions. Full code of linear library is shown in Appendix A.

Code 6.2: Code defining C functions in MLTON

```
val C_createCArr = _import "createCArr" public: int * int-> real array;
val C_toCArr = _import "toCArr" public: real array * int -> real array;
val C_freeVec = _import "freeVec" public : real array -> unit;
val C_printMat = _import "printMat" public : real array * int * int -> unit;
val C_mulVV = _import "mulVV" : int * real array * real array -> real;
val C_mulMV = _import "mulMV" public : int*int * int * real array *
   real array * real array -> unit;
val C_mulMM = _import "mulMM" public : int * int * int * int * real array *
   real array * real array-> unit;
val C_mseCal = _import "mseCal" public : int * int * real array *
   real array * real array * real array * real array -> real;
val C_maeCal = _import "maeCal" public : int * int * real array * real array
   * real array * real array * real array -> real;
val C_solve = _import "solve" public: int * int * real array *
   real array -> real array;
```

Then, a complete function that fit a linear model from data in MLTON is defined as follows in Code 6.3. For a short explanation, this function completely follows the equation 6.5 where A is matrix data and b is the vector of response values. Note that we only take set of attributes that are used in test or splitting below the node, thus A itself contains selected features instead of full data.

Code 6.3: fit a Linear model

```
fun linearModel (usedAtts, dataVals : real list list,
       classVals : real list) =
 let
  val selectedDataVals = selectedAtts(usedAtts, dataVals)
  val nRows = length_of_list(classVals)
  val nCols = length_of_list(get_ith_element_of_list(selectedDataVals,1))
  val selectedData2Mat = fromLists(selectedDataVals)
  val classVals2Vec= Array.fromList classVals
  val A = Array.array(nCols * nCols,0.0)
  val b = Array.array(nCols,0.0)
  val _ = C_mulMM(1,nCols,nRows,nCols,selectedData2Mat,selectedData2Mat, A)
  val _ = C_mulMV(1,nRows,nCols,selectedData2Mat,classVals2Vec, b)
  val N = nCols
  val NRHS = 1
  val _ = C_solve(N,NRHS, A, b)
  val BsmlAr = CArr2SMLArray(b,N)
 in
  (selectedData2Mat, BsmlAr)
 end
```

6.2.2 M5 tree library

A M5 node contains node identification, the attribute position and attribute value it is split, list of attributes that are in split below (to construct the linear model at this node), the values of features X and Y reaching this node. A M5 tree is defined as a group of the current node, left tree and right tree. M5 tree structure is shown in Code 6.4

Code 6.4: Define M5 node and M5 tree structure

```
type M5Node = {
  id : int,
  splitAtt : int,
  splitVal : real,
  usedAtts : bool list,
  classValues : real list,
  dataValues : real list list
}
datatype M5Tree = empty | node of M5Node * M5Tree * M5Tree
```

Basically there are three modules in M5 tree library: growing the tree (splitting nodes), pruning tree and smoothing tree.

Growing M5 tree

Each time M5 tree needs to split a node, it traverses all available features in dataset to choose the best split. For each feature, first it sorts feature values in a ascending manner. In our implementation, we apply quick sort method to optimize the speed of our model. There are $n-1$ possible spit positions in a feature with n different values. For each of them, a reduce in standard deviation of class values is calculated before and after the split then the one with most reduction is selected. An overview of modules implemented to grow a M5 tree is presented in Figure 6.1

To demonstrate, MLTON implementation of *findBestSplit* function is shown in Code 6.5.

Pruning M5 tree

Pruning M5 tree is proceeded in a recursive manner from the leaves to the root. At each node, first a linear model is built taking only attributes that are tested in nodes below. However, to overcome overfitting, terms in the linear model is still remove greedily as long as removing a term does not reduce its performance. In our implementation, we apply the same method with Weka that uses Akaike criterion to grade the goodness of fit of linear models. Error is estimated by Mean Squared Error. A more detailed description of this module is drawn as in Figure 6.2

SML code for M5 tree pruning function is given is Code 6.6.

Code 6.6: M5 pruning function

```
fun prune( tree : M5Tree, paren_linearDict, numAtts : int, initVals)
: int * real * (int * int * real * real array) list * real array * M5Tree =
case tree of
 empty => raise prune_ONEBRANCH
 |node(crrNode,empty,empty) =>
  (
  case findBestLinear(crrNode) of (linear, selectedAtts, selectedDataVals)
  => case coefs2FullLinearCoefs (selectedAtts, linear, numAtts) of
   fullLinearCoefs =>  case length_of_list(#classValues crrNode) of m =>
  case mseCal(m, countSelectedAtts selectedAtts + 1,selectedDataVals,
     linear, Array.fromList (#classValues crrNode)) of error =>
```

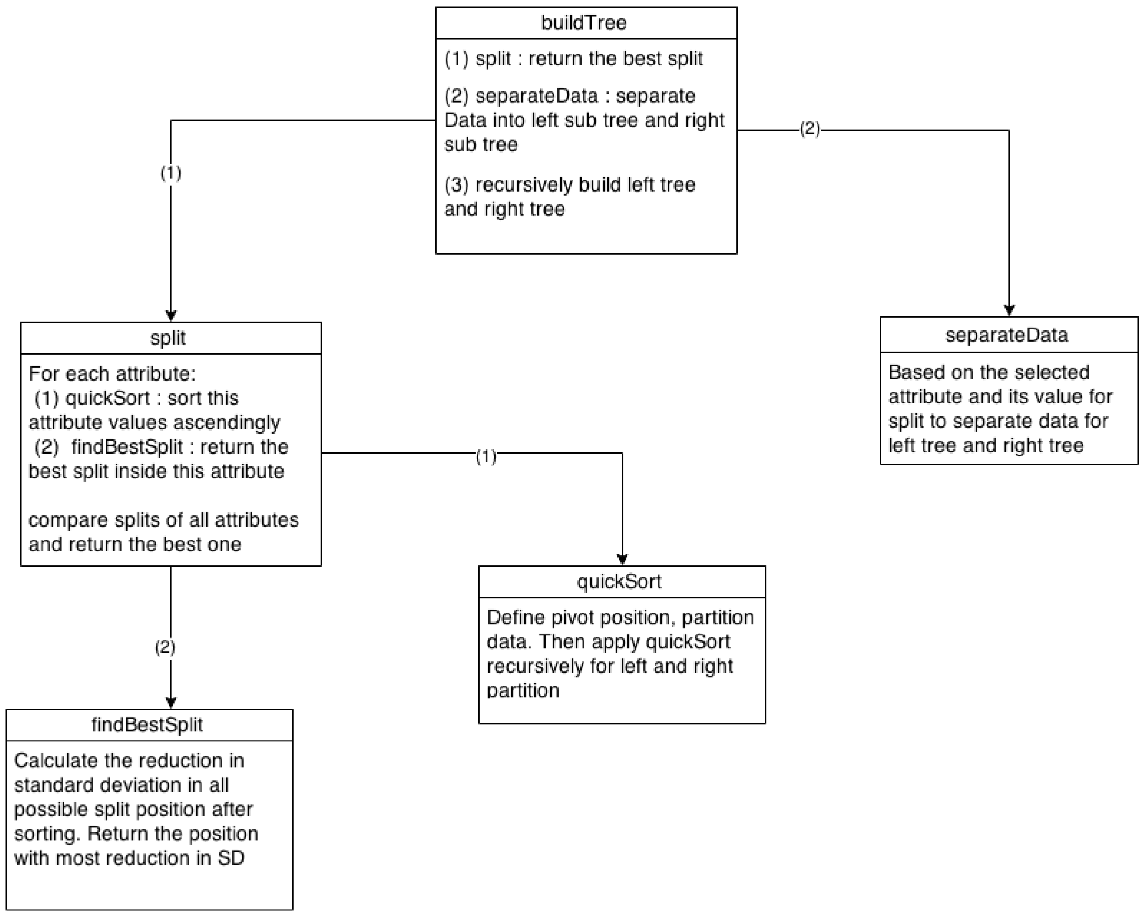

buildTree
(1) split : return the best split
(2) separateData : separate Data into left sub tree and right sub tree
(3) recursively build left tree and right tree
(1)
(2)
split
For each attribute:
(1) quickSort : sort this attribute values ascendingly
(2) findBestSplit : return the best split inside this attribute
compare splits of all attributes and return the best one
(1)
(2)
separateData
Based on the selected attribute and its value for split to separate data for left tree and right tree
quickSort
Define pivot position, partition data. Then apply quickSort recursively for left and right partition
findBestSplit
Calculate the reduction in standard deviation in all possible split position after sorting. Return the position with most reduction in SD

Figure 6.1: Flow chart of M5 tree growing module

Code 6.5: Find best split in an attribute

```
fun findBestSplit(attVals : real array, classVals: real array,
numInstances : int, counter : int, splitIndex, minSD): int * real =
case numInstances - counter of
  0 => (splitIndex, minSD)
  | _ : int =>
   case counter of
    1 =>
         (case  separateArray(1, numInstances, classVals, attVals) of
                (subArray1, subArray2, new_counter) =>
       case (SD_Calculation subArray1) * real(new_counter) +
            (SD_Calculation subArray2) * real(numInstances - new_counter)
            of minSDTemp =>
            case numInstances - new_counter of
             0 => (1, 9999999.0)
             | _ => findBestSplit(attVals, classVals, numInstances,
                   new_counter + 1, new_counter,minSDTemp))
          |_ =>
       (case  separateArray(counter, numInstances, classVals, attVals) of
          (subArray1, subArray2, new_counter) =>
             case numInstances - new_counter of
             0 => (splitIndex, minSD)
             | _ =>
          case (SD_Calculation subArray1) * real(new_counter) +
                 (SD_Calculation subArray2)* real(numInstances- new_counter)
                 of minSDTemp => case minSDTemp < ( minSD - 0.000001) of
                true => findBestSplit(attVals, classVals,numInstances,
                new_counter + 1,  new_counter,minSDTemp)
                |false => findBestSplit(attVals, classVals,numInstances,
                    new_counter + 1, splitIndex,minSD) )
```

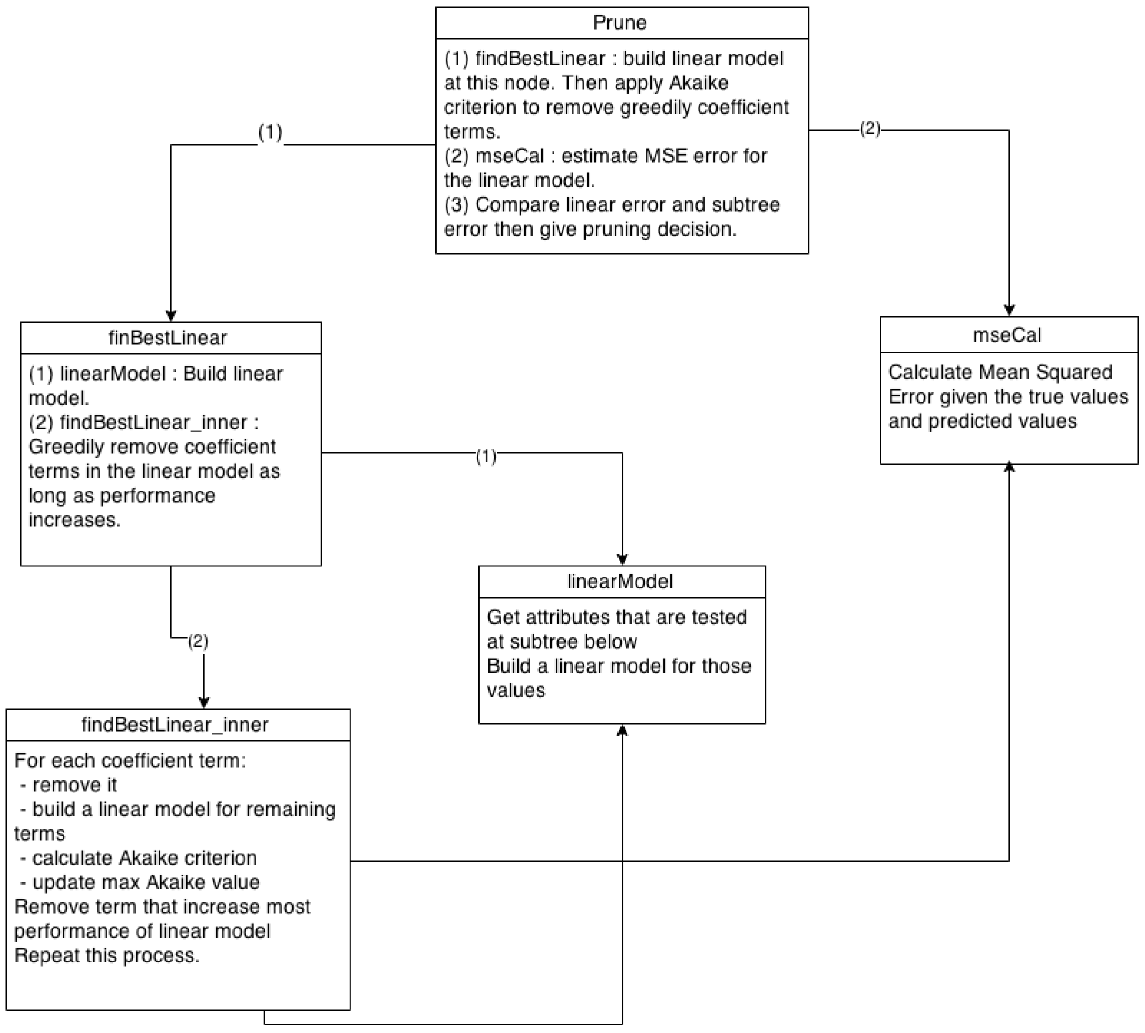

Prune
(1) findBestLinear : build linear model at this node. Then apply Akaike criterion to remove greedily coefficient terms.
(2) mseCal : estimate MSE error for the linear model.
(3) Compare linear error and subtree error then give pruning decision.
(1)
(2)
finBestLinear
(1) linearModel : Build linear model.
(2) findBestLinear_inner : Greedily remove coefficient terms in the linear model as long as performance increases.
(1)
(2)
mseCal
Calculate Mean Squared Error given the true values and predicted values
linearModel
Get attributes that are tested at subtree below
Build a linear model for those values
findBestLinear_inner
For each coefficient term:
- remove it
- build a linear model for remaining terms
- calculate Akaike criterion
- update max Akaike value
Remove term that increase most performance of linear model
Repeat this process.

Figure 6.2: Flow chart of M5 tree pruning module

```
   (1,error ,paren_linearDict , fullLinearCoefs ,tree))

 |node(crrNode, l_Tree, r_Tree) =>

  case prune(l_Tree, paren_linearDict, numAtts, initVals) of
   (l_params, l_error , l_dict , l_linear , l_prunedTree) =>
  case prune(r_Tree ,l_dict , numAtts, initVals) of
    (r_params, r_error , r_dict , r_linear , r_prunedTree) =>

  case findBestLinear(crrNode) of
   (linear, selectedAtts, selectedDataVals) =>
  case coefs2FullLinearCoefs (selectedAtts, linear, numAtts) of
   fullLinearCoefs =>
  case length_of_list(#classValues crrNode) of m =>
  case mseCal(m, countSelectedAtts selectedAtts + 1,  selectedDataVals,
       linear, Array.fromList (#classValues crrNode)) of nodeError =>
  case nodeError * pruneFactor(Real.fromInt m, Real.fromInt
    ( countSelectedAtts selectedAtts + 1), initVals) of adjustedNodeError =>
  case (l_error * (getNumInstances l_Tree) + r_error *
         (getNumInstances r_Tree)) / Real.fromInt m of treeError =>
  case treeError *  pruneFactor(Real.fromInt m, Real.fromInt
       (l_params + r_params + 1), initVals) of adjustedTreeError =>
  case adjustedNodeError > adjustedTreeError of
        true =>
          (l_params + r_params + 1, treeError, (getID l_prunedTree,
           getNumInstances l_prunedTree, l_linear)::(getID r_prunedTree,
           #id crrNode, getNumInstances r_prunedTree, r_linear):: r_dict,
                        fullLinearCoefs, node(crrNode, l_prunedTree,
        |false =>
         (countSelectedAtts selectedAtts + 1,nodeError, r_dict,
              fullLinearCoefs, node(crrNode, empty,empty))
```

Smoothing M5 tree
Smoothing process goes from leaf nodes along the path to the root node. When it comes to a node in the way, coefficient values of the linear model in leaf nodes are updated according to value of that node. Functions implemented for this module is introduced in Figure 6.3

6.2.3 Writing specification file

Specification file is fed to ADATE system and serve as a base to generate evolutionary programs. It is written in ADATE-ML, a subset of Standard ML that is used by ADATE 's inferred programs. The main motivation for using a custom language instead of all of ML is to get a simpler language and one that is purely functional. The main features of ML that are left out of ADATE-ML are higher order functions, fully polymorphic typing, syntactic sugar (extra possibilities to write semantically equal code in different ways) and the module system.

First and foremost, this file contains a *f(.)* function, that is the starting program for ADATE. Code 7.1 show *f(.)* function for smoothing experiments while Code 7.6 show *f(.)* function for pruning experiments.

Then, the specification file loads data in datasets and build trees. Actually ADATE system considers those M5 trees as data. Its operations of training and validating programs are processed on M5 trees. A code example for this is given in Code 6.9

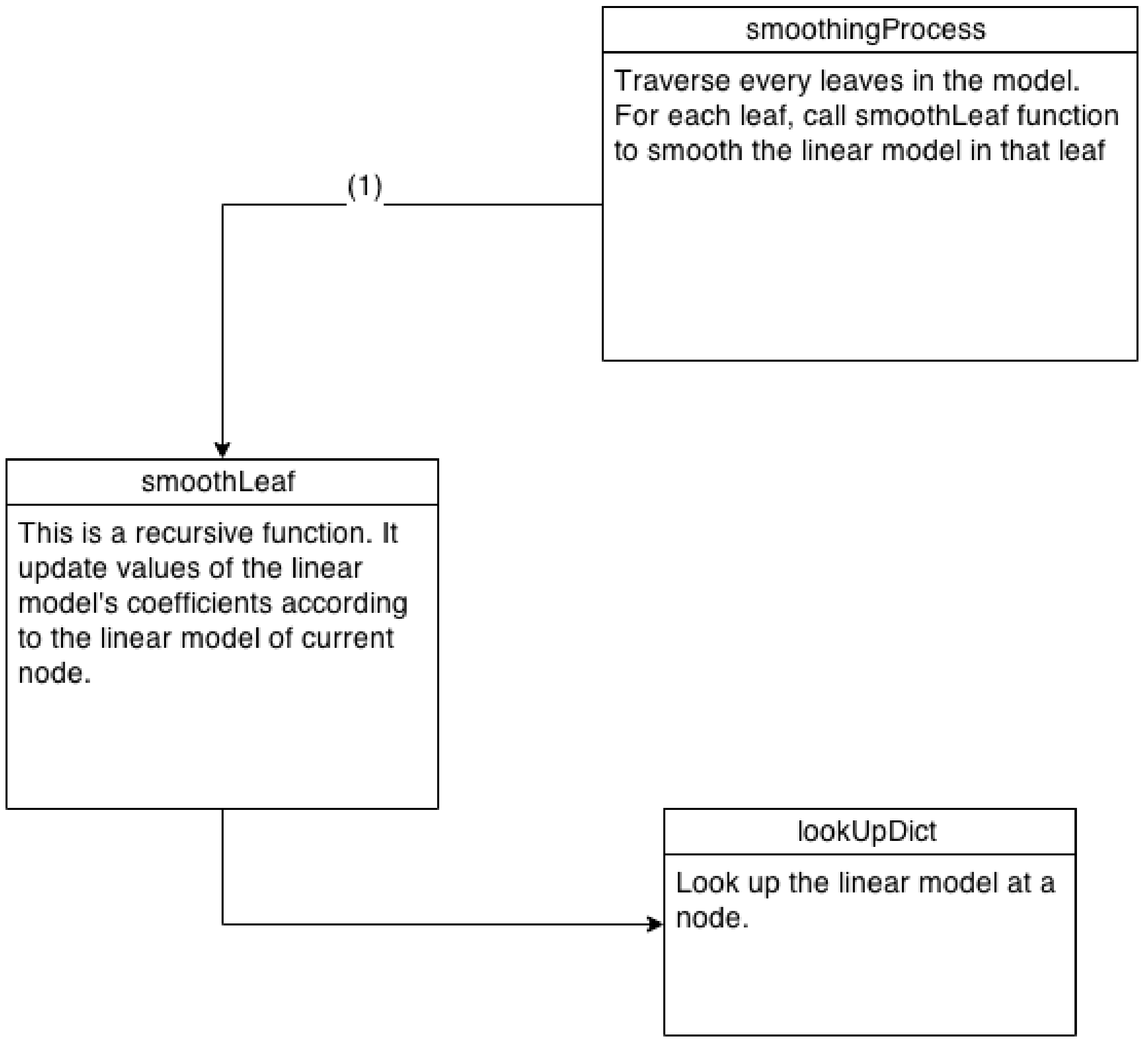

smoothingProcess
Traverse every leaves in the model. For each leaf, call smoothLeaf function to smooth the linear model in that leaf
(1)
smoothLeaf
This is a recursive function. It update values of the linear model's coefficients according to the linear model of current node.
lookUpDict
Look up the linear model at a node.

Figure 6.3: Flow chart of M5 tree smoothing module

Code 6.7: f function for smoothing experiments

```
fun f( ( P, Q, N ) : real * real * real ) : real =
case tor( rconst (0, 5.0, 15.0) ) of K =>
case realMultiply(K,N) of NewK =>
 realDivide(
  realAdd(
   P,
   realMultiply(Q, NewK)
  ),
  realAdd(1.0, NewK)
  )
```

Code 6.8: f function for pruning experiments

```
fun f((E, IN, IV) : real * real * real) : real =
case (realLessOrEqual(IV,IN)) of
 true =>
  realMultiply(
   E,
   10.0
  )
 |false =>
 (
  case  tor( rconst (0, 0.1, 1.0) ) of K =>
  realMultiply(
   E,
   realDivide(
    realAdd(
         IV,
         realMultiply(K, IN)
        ),
    realSubtract(
         IV,
         IN
        )
   )
  )
 )
```

Code 6.9: f function with data loading code

```
(* make dataTrain, dataValid for Inputs - one Fold *)
val allXy = List.map readXy dataset_Inputs
val oneCV_allXy = List.map (oneCVSplit 0.33) allXy
val (dataTrain, dataValid_Inputs) = trainValidSeparate
   (oneCV_allXy, [], [])
val allTrees_Inputs = List.map build_unPrunedTree dataTrain
val allXvalid_Inputs = List.map #1 dataValid_Inputs
val allYvalid_Inputs = List.map #2 dataValid_Inputs

(* make data for Test_inputs - 5 Folds *)
val tree_Xvalid_yvalid_list_list = List.map (treeValidData_nCV (0.2, 5))
   dataset_Test_inputs
val allTrees_Test_inputs = List.concat (List.map #1
   tree_Xvalid_yvalid_list_list)
val allXvalid_Test_inputs = List.concat (List.map #2
   tree_Xvalid_yvalid_list_list)
val allYvalid_Test_inputs = List.concat (List.map #3
   tree_Xvalid_yvalid_list_list)
```

Finally, an output evaluation function is defined to measure performance of programs generated. Performance is calculated on validation data, i.e. the 1/10 fold when remaining 9/10 data is used for training. The selected metric is relative mean squared error (RelMSE), that is mean squared error divided by the variance of data. We choose this metric instead of mean squared error to make the range of errors in interval of $(0, 1)$ and thus eliminate the effect due to the difference of values range from different datasets.

In our implementation of evaluation function, we use a little trick to avoid overfitting. Particularly, we set a threshold for each dataset such that if the error of a newly generated program on the current dataset is less than that threshold, we will set the error to that threshold. This prevents programs from being *so good* that losing its generalized ability. A threshold is estimated as a proportional to the error of original program on each dataset. This segment of code is shown in Code 6.10

Code 6.10: Avoid overfitting in evaluation function

```
val initError = List.nth( initRelMSE, I)
  val error =
    if I >= N then
      ActualError
    else if ActualError < ( 1.0 - Threshold ) * initError then
      ( 1.0 - Threshold ) * initError
    else
      ActualError
```

Datasets	Instances	Missing values (%)	Numeric attributes	Nominal attributes
2D Planes	40768	0.0	10	0
Abalone	4177	0.0	7	1
Add10	9792	0.0	10	0
Ailerons	13750	0.0	40	0
bank8FM	8192	0.0	8	0
bank32nh	8192	0.0	32	0
California House	20640	0.0	8	0
CASP	45730	0.0	9	0
CBM	11934	0.0	17	0
Combined Cycle Power Plant	9568	0.0	5	0
CPU (small)	8192	0.0	9	3
Delta Ailerons	7129	0.0	5	0
Delta Ail	7129	0.0	6	0
Anacalt	4052	0.0	1	6
CPU Act	8192	0.0	19	2
Delta Elevator	9517	0.0	5	0
Relation Network Directed	53413	0.0	22	0
Elevators	16599	0.0	18	0
Fried Delve	40768	0.0	10	0
housePrice8	22784	0.0	8	0
housePrice16	22784	0.0	16	0
hwang	13600	0.0	11	0
Kinematics8nm	8192	0.0	8	0
MV artificial	40768	0.0	7	3
parkinsons_updrs	5875	0.0	19	2
Pole48	15000	0.0	48	0
Pole Telecom	14998	0.0	26	0
puma8NH	8192	0.0	8	0
puma32H	8192	0.0	32	0
Wine Quality	4898	0.0	11	0

Table 6.1: Overview description of 30 datasets in ADATE experiments

	Root MSE	Relative Root MSE
M5 Prime	**58829.80**	**0.51**
Bagging	52904.87	0.458
M5 Rules	55784.75	0.483
Additive Regression	57645.78	0.499

Table 6.2: A comparison of different learners on California House dataset

Chapter 7

ADATE experiments: Results and Discussion

In this chapter, result of experiments described in Chapter 6 are presented and analyzed. Results include synthesized programs for each target and their performances in a comparison to the original expression used in M5 algorithm. We apply Relative Mean Squared Error (RelMSE) as a measure of programs' performance.

7.1 M5 smoothing improvement experiments

Aim for modifying the expression employed in smoothing procedure,

$$p' = \frac{np + kq}{n + k} \tag{7.1}$$

we start this experiment by a program that represents the expression 7.1, as given in Code 7.1.

Code 7.1: Starting program for smoothing experiment

```
fun f( ( P, Q, N ) : real * real * real ) : real =
case tor( rconst (0, 5.0, 15.0) ) of K =>
 realDivide(
  realAdd(
   realMultiply(P, N),
   realMultiply(Q, K)
  ),
  realAdd(N, K)
  )
```

7.1.1 M5 smoothing on California House dataset

Recall that in this experiment, we divide the dataset into 10 folds. For each fold, there is a M5 tree built for 9/10 data and its performance is estimated on the remaining 1/10 data. However these M5 trees are not smoothed, instead they waits for programs generated by ADATE to smooth later. Performance of a program is estimated by applying a tree with new smoothing function on validation set. ADATE trains its model with the first 5 trees while preserving the last 5 trees for validation. Two of programs generated by ADATE, *f1* and *f2*, are given in Code 7.2 and Code 7.3. While *f2* program is the best program that ADATE can synthesize in the pre-determined amount of time, *f1* program is picked randomly along the evolutionary process so we can have a broader perspective of how ADATE works.

Code 7.2: f1 program generated for smoothing experiment on California House

```
fun f1( P, Q, N ) =
  case
    case rconstLess( N, rconst( 0, 2.5, 4.549819233 ) ) of
      false => tor( rconst( 11, 0.402491214607, 20.0199634135 ) )
    | true =>
        realSubtract( tor( rconst( 0, 2.5, 3.52146578646 ) ), N )   of
    V100E3 =>
      realDivide(
        realAdd( realMultiply( P, N ), realMultiply( Q, V100E3 ) ),
        realAdd( N, V100E3 )
        )
```

It is obvious that *f2* is more complicated than *f1*. The results validated on California House also shows that *f2* performs better than *f1*. Further more, *f1* is actually worse than original *f* program. This fact can be explained by the way we conduct this experiment that only 5 out of 10 trees are used to train ADATE, thus *f1* might not be generalized well when it is tested on all 10 folds of this dataset. To see how these two programs perform on other datasets, we have tested them on remaining 29 datasets, all with 10 folds cross validation, the result is presented in Table 7.1.

From Table 7.1, *f1* performs better than original *f* program at 15 out of 30 datasets, however it is worse than *f* program for overall score, with a sum of RelMSE over 30 datasets being 12.2850 compared to 12.2838. On the other hands, *f2* program beats *f* program for 17 datasets and yields a better overall score than *f*, with a RelMSE being 12.27321646. For the California House dataset that is used to train ADATE in this experiment, the program $f2$ can improve performance of M5 tree up to 0.2%. Among all datasets, *f2* offers the most increase in predictive accuracy for CASP with 0.98%. *f2* also generalize quite well on other datasets ,although the results are not too significant.

7.1.2 M5 smoothing on 30 datasets - one fold

Different from the first experiment, in this case we were looking for a more stable and generalized program that hopefully can improve the ”quality” of smoothing process on not only one but many datasets. To achieve that, we fit data from all 30 datasets to ADATE, 20 of them for training and 10 for validating. Also, for each dataset we separate only one fold, with training set over testing set ration being 66% : 33%, for the purpose of optimizing ADATE training time. Here we present Code 7.4 as the best program ADATE could come up with.

As a result, generally *f6* offers a higher performance than *f* on the same testing sets (each dataset is split with ratio 66% : 33% into training set and testing set). In Table 7.2, we can see that the overall relative mean squared error of *f* and *f6* for 1-fold validation are 12.40057527 and 12.31773707 respectively. Among all datasets, *f6* significantly increase the accuracy of M5 model tree up to 7% on CASP dataset.

Code 7.3: f2 program generated for smoothing experiment on California House

```
fun f2( P, Q, N ) =
  case realDivide( tor( rconst( 0, 5.0, 15.0 ) ), N ) of
    K =>
  case
    Math.tanh(
      realAdd(
        realAdd(
          realAdd(
            tor( rconst( 0, 0.25, 0.360557744994 ) ),
            case
              case
                realDivide(
                  K,
                  realSubtract(
                    tor( rconst( 0, 0.25, 0.233256970613 ) ),
                    K
                    )
                  )                 of
                V207F0 => realAdd( V207F0, V207F0 ) of
              V1BD3F => realMultiply( V1BD3F, V1BD3F )
            ),
          K
          ),
        K
        )
      )    of
    V2E3 =>
      realDivide(
        realAdd( realMultiply( P, V2E3 ), realMultiply( Q, K ) ),
        realAdd( V2E3, K )
        )
```

Code 7.4: f6 program generated for smoothing experiment on 30 datasets- one fold

```
fun f6( P, Q, N ) =
  case
    rconstLess( N, rconst( 5, 0.03156328125, 0.151467238993 ) )    of
      false => Q
  | true =>
  case
    case rconstLess( N, rconst( 0, 0.25, 0.107651670053 ) ) of
      false => N
    | true =>
        realMultiply(
          tor( rconst( 39, 0.7949666685788486E~7, 18.4730153162 ) ),
          N
          )    of
    V7AA56 =>
      realDivide(
        realAdd( P, realMultiply( Q, V7AA56 ) ),
        realAdd( 1.0, V7AA56 )
        )
```

However, when we check the overfiting possibility of *f6* by testing on 10-folds cross validation, Table 7.2 shows that it is not as good as original *f* program. More particularly, *f6* scores an overall RelMSE of 12.3019379, higher than *f* with 12.28380234. So we can conclude that using only one fold of datasets to train ADATE system causes programs generated by ADATE overfitted, or in other words loose its generalization ability.

7.1.3 M5 smoothing on 30 datasets - five folds

We have seen that fitting only one tree each dataset to ADATE makes generalization ability of ADATE programs worse, as given in finding in experiment number 2 above. Therefore in the experiment number 3, we hope to overcome this problem by increase the number of trees each dataset to 5. Theoretically, more training data means more patterns are captured and it increases the chance ADATE programs could perform well on testing data. And actually the result of this experiment proves our hypothesis is right. Program *f10*, given in Code 7.5, scores better relative mean squared error than *f*.

Code 7.5: program *f10* generated for smoothing experiment on 30 datasets- five folds

```
fun f( P, Q, N ) =
 case
  realAdd(
   N,
   realMultiply( tor( rconst( 2, 2.5125, 20.025)) tanh(N))
     )   of
    NewK =>
      realDivide(
        realAdd( P, realMultiply( Q, NewK ) ),
        realAdd( 1.0, NewK )
        )
```

Out of 30 datasets, *f10* gives higher performance than *f* on 19 datasets, especially highest improvement for housePrice16 with an increase of 1.62%. We calculate the overall relative mean squared error of program *f10* is 12.26619365, 1.76% lower than this number for *f*. Details for this experiment' s results are presented in Table 7.1.3

To summarize this section of M5 smoothing experiments, *f10* is the best program generate by ADATE to alter the original expression used in smoothing process 7.1. It can improve the performance of M5 up to 1.76% on 30 tested datasets.

7.2 M5 pruning improvement experiments

For pruning experiments, our target is to find alternative programs rather than $(n+v)/(n-v)$ for estimating error on unseen cases from training error. To start, the input *f* program representing this expression is fed to ADATE system. The SML starting program is shown in Code 7.6.

Similar to experiment number 3 as in smoothing, here we also use all 30 datasets divided into 5 folds. One minor difference is that we replace Electricity Boards dataset by Relation Network Directed dataset because the effect of M5 learner on Electricity Boards is not much. As a result, *f14* is the best program synthesized during the evolutionary process. *f14* is given in Code 7.7.

Code 7.6: f stating program for pruning experiments

```
fun f((E, IN, IV) : real * real * real) : real =
case (realLessOrEqual(IV,IN)) of
 true =>
  realMultiply(
   E,
   10.0
  )
 |false =>
 (
  case  tor( rconst (0, 0.1, 1.0) ) of K =>
  realMultiply(
   E,
   realDivide(
    realAdd(
         IV,
         realMultiply(K, IN)
        ),
    realSubtract(
         IV,
         IN
        )
   )
  )
 )
```

Code 7.7: best program for pruning experiment

```
fun f4( E, IN, IV ) =
 case realLessOrEqual( IV, IN ) of
  false =>
   realMultiply(
    E,
    realDivide(
     realAdd(
      IV,
      realMultiply(
       case
        rconstLess(
         IV,
         rconst( 0, 0.25, 0.117780139933 )
           )                of
        false => IV
         | true =>
         realAdd( tor( rconst( 1, 0.1005, 1.201 ) ), IV ),
          IN
          )
        ),
      Math.tanh( realSubtract( IV, IN ) )
    )
        )
 | true =>
  realMultiply( IN, tor( rconst( 0, 25.0, 27.7810244153 ) ) )
```

10-folds cross validation results of program *f14* are presented in Table 7.4. It significantly improve performance of M5 tree on parkinsons dataset with an increase of 2.36%. Totally for all datasets, *f14* yields a relative mean squared error of 11.51642588, better than original program *f* with 11.56851245. An improvement of 5.21% over 30 datasets is really an encouraging result for us.

Dataset	**RelMSE f**	**RelMSE f1**	**RelMSE f2**
2DPlanes	0.227161553	0.227161522	0.227162688
Abalone	0.656460111	0.656939107	0.655382643
Add10	0.382324554	0.384503829	0.384701841
Ailerons	0.397625081	0.397506567	0.39728005
bank8FM	0.200856563	0.200639324	0.20078268
bank32nh	0.680085028	0.679707724	0.678022783
California House	0.471389635	0.477965165	0.469703371
CASP	0.742078189	0.741549541	0.732261045
CBM	0.020538716	0.019144381	0.022579367
Combined Cycle Power Plant	0.223177903	0.222847986	0.223063674
CPU (small)	0.168704594	0.170095243	0.169367971
Delta Ailerons	0.541864674	0.543569906	0.540750576
Delta Ail	0.541864674	0.543569906	0.540750576
Anacalt	0.147834494	0.147492776	0.150025514
CPU Act	0.146842842	0.146053296	0.145742638
Delta Elevator	0.599225594	0.599034811	0.598898679
ElectricityBoards 45781	1.000527406	1.000527406	1.000527406
Elevators	0.326046226	0.325773621	0.324953257
Fried Delve	0.296136539	0.299393719	0.300237193
housePrice8	0.675514062	0.678325835	0.67206411
housePrice16	0.624965699	0.603060349	0.616811994
hwang	0.243564421	0.24356442	0.24356442
Kinematics8nm	0.641145917	0.642893848	0.641975142
MV artificial	0.01611187	0.016180068	0.01681686
parkinsons	0.402836996	0.402652005	0.402502403
Pole48	0.128566662	0.129874572	0.130961437
Pole Telecom	0.12535655	0.126034034	0.128036381
puma8NH	0.573613771	0.57354215	0.574000948
puma32H	0.282301321	0.283485084	0.28844738
Wine Quality	0.7990807	0.801956278	0.795841432
Sum RelMSE	**12.28380234**	**12.28504447**	**12.27321646**

Table 7.1: Results of *f1* and *f2* programs on 30 datasets

Dataset	**10-fold RelMSE**		**1-Fold RelMSE**	
	f	**f6**	**f**	**f6**
2DPlanes	0.227161553	0.227161521	0.225507374	0.225505361
Abalone	0.656460111	0.655865296	0.672864009	0.683176672
Add10	0.382324554	0.383464231	0.39935063	0.401003246
Ailerons	0.397625081	0.397541655	0.398175898	0.397942079
bank8FM	0.200856563	0.200815488	0.196253619	0.19608286
bank32nh	0.680085028	0.679093716	0.690485416	0.689587774
California House	0.471389635	0.472194748	0.473354878	0.47367005
CASP	0.742078189	0.746160788	0.841869122	0.771423114
CBM	0.020538716	0.019941859	0.011341854	0.012236432
Combined Cycle Power Plant	0.223177903	0.229225919	0.231948255	0.232598596
CPU (small)	0.168704594	0.170471412	0.167757819	0.169789544
Delta Ailerons	0.541864674	0.541182197	0.53138945	0.529307517
Delta Ail	0.541864674	0.541182197	0.53138945	0.529307517
Anacalt	0.147834494	0.150617786	0.126668861	0.127428519
CPU Act	0.146842842	0.147867509	0.138701062	0.140345635
Delta Elevator	0.599225594	0.599050555	0.60336591	0.605169481
ElectricityBoards 45781	1.000527406	1.000527406	1.000331157	1.000331157
Elevators	0.326046226	0.32556866	0.33170983	0.331617126
Fried Delve	0.296136539	0.298250616	0.307437999	0.309687432
housePrice8	0.675514062	0.675731452	0.65624039	0.65089561
housePrice16	0.624965699	0.624217147	0.565388057	0.558752692
hwang	0.243564421	0.243564421	0.246722836	0.246722836
Kinematics8nm	0.641145917	0.641850515	0.665371991	0.6668078
MV artificial	0.01611187	0.016156174	0.020745204	0.020810186
parkinsons	0.402836996	0.402701654	0.413909074	0.413584302
Pole48	0.128566662	0.13048015	0.130860254	0.132594752
Pole Telecom	0.12535655	0.127247584	0.143534395	0.139762772
puma8NH	0.573613771	0.573607746	0.569334875	0.569181979
puma32H	0.282301321	0.282524597	0.287756993	0.289087637
Wine Quality	0.7990807	0.797672909	0.820808609	0.803326396
Sum RelMSE	**12.28380234**	**12.3019379**	**12.40057527**	**12.31773707**

Table 7.2: Results of *f6* on 30 datasets for 1-fold and 10-fold cross validation

Dataset	**10-folds RelMSE f**	**10-folds RelMSE f10**
2DPlanes	0.227161553	0.227161528
Abalone	0.656460111	0.655592868
Add10	0.382324554	0.384846306
Ailerons	0.397625081	0.397481412
bank8FM	0.200856563	0.200729179
bank32nh	0.680085028	0.6778643
California House	0.471389635	0.470022093
CASP	0.742078189	0.744579252
CBM	0.020538716	0.0189431
Combined Cycle Power Plant	0.223177903	0.223072125
CPU (small)	0.168704594	0.17011096
Delta Ailerons	0.541864674	0.540676429
Delta Ail	0.541864674	0.540676429
Anacalt	0.147834494	0.147969192
CPU Act	0.146842842	0.145835501
Delta Elevator	0.599225594	0.599005296
ElectricityBoards 45781	1.000527406	1.000527406
Elevators	0.326046226	0.325110807
Fried Delve	0.296136539	0.300005511
housePrice8	0.675514062	0.669793896
housePrice16	0.624965699	0.608727861
hwang	0.243564421	0.24356442
Kinematics8nm	0.641145917	0.642474146
MV artificial	0.01611187	0.016196949
parkinsons	0.402836996	0.402612482
Pole48	0.128566662	0.130234177
Pole Telecom	0.12535655	0.127324226
puma8NH	0.573613771	0.573541191
puma32H	0.282301321	0.282207372
Wine Quality	0.7990807	0.799307233
Sum RelMSE	**12.28380234**	**12.26619365**

Table 7.3: Results of *f10* on 30 datasets for 10-fold cross validation

Dataset	**10-folds RelMSE f**	**10-folds RelMSE f14**
2DPlanes	0.227121408	0.227838771
Abalone	0.669706988	0.66420064
Add10	0.36156611	0.368688568
Ailerons	0.397763234	0.394519557
bank8FM	0.200462336	0.2003889
bank32nh	0.733198778	0.706808234
California House	0.451229745	0.464509504
CASP	0.694658551	0.70223474
CBM	0.020560585	0.01606437
Combined Cycle Power Plant	0.300532968	0.301866582
CPU (small)	0.179842365	0.168277599
Delta Ailerons	0.54267631	0.543610214
Delta Ail	0.54267631	0.543610214
Anacalt	0.146898824	0.146774375
CPU Act	0.147100802	0.140928582
Delta Elevator	0.608727126	0.602588809
Relation Networks Directed	0.22183107	0.223593793
Elevators	0.331398932	0.330942546
Fried Delve	0.288916483	0.289807517
housePrice8	0.666104582	0.67197361
housePrice16	0.621426042	0.606766916
hwang	0.243458239	0.246223521
Kinematics8nm	0.62692933	0.631133997
MV artificial	0.016111866	0.016144164
parkinsons	0.421842926	0.398183732
Pole48	0.129041877	0.129302183
Pole Telecom	0.123711657	0.124248905
puma8NH	0.577676281	0.577364167
puma32H	0.27751388	0.276696389
Wine Quality	0.797826842	0.801134785
Sum RelMSE	**11.56851245**	**11.51642588**

Table 7.4: 10-folds cross validation results of the best program by ADATE, *f6*, on 30 datasets

Chapter 8

Conclusion and Future Works

8.1 Conclusion

The contribution of this dissertation can be summarized into two main aspects: (i) we have investigated decision tree learners, particularly focus on M5 model tree and its related models; (ii) we conducted experiments using ADATE to improve M5 model tree algorithm.

We have presented task (i) in Chapters from 2 to 4. We first recognize the power of decision tree learners in a wide range of Machine Learning problems. Then we take a deeper look insight the structure of classification and regression trees, the way strategy *Divide and Conquer* works and how to avoid overfitting that is actually a serious problem in tree models. We debate that ensemble of trees is a good trees-related method that usually yield a better performance than individuals and helpful in tackling overfitting. Finally with this purpose, we introduce M5 model tree, a state-of-the-art model among decision tree learners. Three main parts of M5 tree, which are growing, pruning and smoothing a tree, are investigated along with M5-based modified algorithms. This knowledge is useful in selecting targets for experiment in the next task.

In terms of task (ii), we start by a research on ADATE system, a system that employs automatic programming and evolutionary algorithm to generate solutions given a starting program and an output evaluation function. ADATE can apply in both classification and regression problems, also can be used to solve special issues as imbalanced target variable. The main focus of this part is to answer the research question, "How and to what extent can ADATE system improve M5 model tree?". We choose to improve smoothing and pruning process in a number of experiments. During the implementation of experiments, two topics need to be taken into consideration are to optimize the running time of f functions and to deal with overfitting. In our case, the overfitting problem is raised in two forms. First when a program could be good on specific parts of data but bad on the other parts, this happens in smoothing experiment number 1. Secondly, a program might performs well on a dataset but does not generalize well on the other datasets.

Findings from experiments show that ADATE system can improve M5 model tree, for both aspects of smoothing and pruning procedure. While *f910* is the best program generated by ADATE to replace the expression in smoothing with an increase in accuracy being 1.76%, *f14* can significantly improve performance of M5 trees when used to alter the testing estimation expression in pruning. M5 tree with *f14* can boost the accuracy up to 5.21% over 30 datasets .Given the size of starting programs and a limited running time of ADATE, those results is very encouraging and prove the potential power of ADATE system.

8.2 Future works

For further research and experiments based on the work done in this project, we suggest several following topics:

- *Enlarge f function in ADATE experiments:* It can be seen that size of starting programs in our experiments are pretty small. It comes from the fact that we only want to change a very small part in M5 algorithm and still remain its nature as well as its main ideas. As a consequence, the effect of new programs on M5 tree is not very significant. We wonder how if we feed bigger programs that cover broader parts of this algorithm. This is also a good chance to observe closer the evolutionary ability of ADATE. An example for this idea is to bring the whole smoothing or pruning function into ADATE to change. This possibility could be done because as far as I know, a recently updated version of ADATE can support to make function calls from environment outside of ADATE-ML, which is C in our case.

- *Improve growing M5 tree process:* In our opinion, this is really an interesting experiment to do. Unlike pruning or smoothing processes, changing the way M5 tree grown can lead to a completely different kind of tree and it might offer many possibilities we never expect. A possibility is to alter the criteria when splitting node, instead of using standard deviation. Also, splitting node process currently involves only one variable at a time and we might would like to include more, i.e a linear expression of variables.

- *An ensemble of M5 trees:* Mention to this direction of research, we would like to put effort into an ensemble model like Random Forest. But instead of using Random tree, we will combine M5 trees together. We believe this new model could perform well on regression problems, at least as well as M5 tree. Of course we need to verify this idea first before conducting experiments related to ADATE, however it is still worth trying in our opinion.

Bibliography

[1] 1993 new car data. www.amstat.org/publications/jse/v1n1/datasets.lock.html.

[2] Adate system. http://www-ia.hiof.no/~rolando/.

[3] Amazon access employee competition. https://www.kaggle.com/c/amazon-employee-access-challenge.

[4] Blas. http://www.netlib.org/blas/.

[5] C5.0. https://www.rulequest.com/see5-unix.html.

[6] Face completion with a multi-output estimators. http://scikit-learn.org/stable/auto_examples/plot_multioutput_face_completion.html#example-plot-multioutput-face-completion-py.

[7] Iris data set. https://archive.ics.uci.edu/ml/datasets/Iris.

[8] Mlton. http://mlton.org/.

[9] Openblas. http://www.openblas.net/.

[10] Uci repository. https://archive.ics.uci.edu/ml/datasets.htmll.

[11] A. Biermann. Automatic programming. *In Encyclopedia of Artificial Intelligence*, pages 18–35, 1992.

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

[13] Leo Breiman. Technical note: Some properties of splitting criteria. *Machine Learning*, 24(1):41–47, 1996.

[14] B. Cestnik and I. Bratko. On estimating probabilities in tree pruning. pages 138–150, 1991.

[15] Houtao Deng and George Runger. Feature selection via regularized trees. *The 2012 International Joint Conference on Neural Networks (IJCNN)*, 2012.

[16] Tang S Doksum K and Tsui KW. *Nonparametric variable selection: the EARTH algorithm*. 2008.

[17] Stig erland Hansen and Roland Olsson. Improving decision tree pruning through automatic programming.

[18] E. Frank, Y. Wang, S. Inglis, G. Holmes, and I.H. Witten. Using model trees for classification. *Machine Learning*, 32(1):63–76, 1998.

[19] Eibe Frank. *Pruning Decision Trees and Lists.* PhD thesis, University of Waikato, 2000.

[20] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[21] M. Hall G. Holmes and E. Frank. Generating rule sets from model trees. In *in Proc. of the 12th Australian Joint Conf. on Artificial Intelligence*, pages 1–12. Springer-Verlag.

[22] Kass GV. *An exploratory technique for investigating large quantities of categorical data.* 1980.

[23] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[24] O'Muircheartaigh CA HFielding A. Binary segmentation in survey analysis with particular reference to aid. *The Statistician*, 1977.

[25] Dang Ha The Hien. *Improving Deep Learning through Automatic Programming.* PhD thesis, 2014.

[26] A. Karalic and B. Cestnik. The bayesian approach to tree-structured regression. *Proceedings of ITI-91*, 1991.

[27] Loh WY. Kim H. *Classification trees with bivariatelinear discriminant node models.* 2003.

[28] Wei-Yin Loh. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23, 2011.

[29] Hordle W Unwin A Loh WY, Chen C. *Improving the precision of classification trees.* 2009.

[30] Shih Y. Loh WY. *Split selection methods for classification trees.* 1997.

[31] Vanichsetakul N. Loh WY. *Tree-structured classificationvia generalized discriminant analysis (with discussion).* J Am Stat Assoc, 1988.

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

[33] Tom M. Mitchell. *Machine Learning.* 1997.

[34] T. Niblett and I. Bratko. Learning decision rules in noisy domains. pages 24–25, 1986.

[35] J. Roland Olsson. The art of writing specifications for the ADATE automatic programming system. In *Proceedings of the Annual Genetic Programming Conference*, pages 278–283. Morgan Kaufmann, 1998.

[36] Roland Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74:55–81, 1995.

[37] Roland Olsson. How to invent functions. volume 1598 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1999.

[38] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 1992.

[39] J. R. Quinlan. Induction of decision trees. machine learning. pages 81–106, 1986.

[40] J. R. Quinlan. Learning with continuous classes. pages 343–348. World Scientific, 1992.

[41] Chotirat Ann Ratanamahatana and Dimitrios Gunopulos. Scaling up the naive bayesian classifier: Using decision trees for feature selection.

[42] C. Rich and R. C. Waters. Approaches to automatic programming. *MITSUBISHI ELECTRIC RESEARCH LABORATORIES*, 1992.

[43] L. F. R. A. Torgo. *Inductive Learning of Tree-based Regression Models.* PhD thesis, 1999.

[44] Yong Wang and Ian H. Witten. Inducing model trees for continuous classes. In *In Proc. of the 9th European Conf. on Machine Learning Poster Papers*, pages 128–137, 1997.

[45] Yong Wang and Ian H. Witten. Inducing model trees for continuous classes. In *In Proc. of the 9th European Conf. on Machine Learning Poster Papers*, pages 128–137, 1997.

[46] Yong Wang and Ian H. Witten. Inducing model trees for continuous classes. In *In Proc. of the 9th European Conf. on Machine Learning Poster Papers*, pages 128–137, 1997.

[47] C. Wild and G Weber. *Introduction to Probability and Statistics.* University of Auckland, 1995).

Appendix A

Linear Regression Library

A.1 Matrix - Vector operations

```
#include "OpenBLAS/cblas.h"
#include "OpenBLAS/common.h"
#include "export.h"
#include "stdio.h"
#include <math.h>
Pointer createCArr(int l, int init){
        double *a = malloc(l * sizeof(double));
        if (init == 1){
         int i =0;
         for (i = 0; i < l; i++)
          a[i] = 0.0;
        }
        return &a[0];
}

Pointer toCArr (double* x, int l){
        double *y = malloc( l * sizeof(double) );
        memcpy(&y[0], &x[0], l * sizeof(double) );
        return &y[0];
}
void freeVec (double* vec){
        free(vec);
}


void printMat (double* x, int m, int n){
        int i, j = 0;

        for (i = 0; i < m; i = i + 1){
                for (j = 0; j < n; j = j + 1){
                        printf("%f ", x[i*n + j]);
                }
                printf("\n");
        }
}
void printVec ( double* x, int l){
        printMat (&x[0], 1, l);
}
int numNotZeroCoef (double* x, int n) {
        int i =0;
        int count = 0;
        for (i = 0; i< n; i = i + 1){
                if (x[i] != 0) count = count + 1;
        }
        return count;
}
double getElement (double* x, int index) {
        return x[index];
}
double mseCal(int m, int n,double* dataVals,double* coeffs,
        double* classVals, double* preds, double* diff){
   cblas_dgemv( CblasRowMajor, CblasNoTrans, m, n, 1, &dataVals[0], n,
               &coeffs[0], 1, 0, &preds[0], 1 );

  cblas_daxpy(m,-1.0,classVals,1,diff,1);
  cblas_daxpy(m,1.0,preds,1,diff,1);
  double mse= cblas_dnrm2(m,diff,1) / sqrt(m);
  return mse;
}
double maeCal(int m, int n,double* dataVals,double* coeffs,
              double* classVals, double* preds, double* diff){
   cblas_dgemv( CblasRowMajor, CblasNoTrans, m, n, 1, &dataVals[0], n,
                &coeffs[0], 1, 0, &preds[0], 1 );
  cblas_daxpy(m,-1.0,classVals,1,diff,1);
  cblas_daxpy(m,1.0,preds,1,diff,1);
  double mae= cblas_dasum(m,diff,1) / m;
  return mae;
}
void mulMM (int transA ,int m,int k, int n, double* A, double *B,
            double* res){
  if (transA == 1)
   cblas_dgemm(CblasRowMajor,CblasTrans,CblasNoTrans,m,n,k,1.0,&A[0],
              m,&B[0],n, 0.0,&res[0],n);
  else
   cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,m,n,k,1.0,&A[0],
               k,&B[0], n,0.0,&res[0],n);
}

double mulVV (int n, double * x, double *y) {
 double res = cblas_ddot (n, &x[0], 1, &y[0], 1);
 return res;
}

void mulMV (int tran, int pm, int pn, double* pa, double* px, double* y)
{
   double alpha, beta;
   int m, n, lda, incx, incy, i;

   m = pm; /* Size of Column ( the number of rows ) */
   n = pn; /* Size of Row ( the number of columns ) */
   lda = n; /* Leading dimension of 5 * 4 matrix is 5 */
   incx = 1; // fixed 1
   incy = 1;
   alpha = 1;
   beta = 0;

   if ( tran == 1)
```

```
    cblas_dgemv( CblasRowMajor, CblasTrans, m, n, alpha, &pa[0], lda,
                 &px[0], incx, beta, &y[0], incy );
    else
    cblas_dgemv( CblasRowMajor, CblasNoTrans, m, n, alpha, &pa[0], lda,
                 &px[0], incx, beta, &y[0], incy );
}
 double iden( double x)
{
 return x * x;
}
```

A.2 Solve Matrix operation

```
#include "OpenBLAS/lapack-netlib/lapacke/include/lapacke.h"
#include "export.h"
#include "stdio.h"
void print_matrix( char* desc, int m, int n, double* a, int lda ) {
        int i, j;
        printf( "\n %s\n", desc );
        for( i = 0; i < m; i++ ) {
                for( j = 0; j < n; j++ ) printf( " %6.2f", a[i+j*lda] );
                printf( "\n" );
        }
}

/* Auxiliary routine: printing a vector of integers */
void print_int_vector( char* desc, int n, int* a ) {
        int j;
        printf( "\n %s\n", desc );
        for( j = 0; j < n; j++ ) printf( " %6i", a[j] );
        printf( "\n" );
}
void Row2Col( int m, int n, double* a, double* res ) {
        int i, j;
        for( i = 0; i < m; i++ ) {
                for( j = 0; j < n; j++ ) res[i+j*m] = a[i*n+j] ;

        }
}
void Col2Row( int m, int n, double* a, double* res ) {
        int i, j;
        for( i = 0; i < m; i++ ) {
                for( j = 0; j < n; j++ ) res[i*n+j] = a[i+j*m] ;

        }
}
//Solve AX= B
void solve(int N, int NRHS, double* A, double* B) {
        //fprintf(stderr, "solve 1 \n");
    int n = N, nrhs = NRHS, lda = N, ldb = NRHS, info;
    int ipiv[N];
        int success = 0;
        double ridge = 10^(-8);
        int i ,j;
        while (success == 0){

                info = LAPACKE_dgesv( LAPACK_ROW_MAJOR,n, nrhs, &A[0], lda,
                        &ipiv[0], &B[0], ldb );
                /* Check for the exact singularity */
                if( info > 0 ) {
                        // exit( 1 );
                        for (i = 0; i < n; i++){
                                A [i*n + i] = A[i*n + i] + ridge;
                        }
                        ridge = ridge * 10.0;
                }
                else
                        success = 1;
        }
}
```

Appendix B

M5 Model Tree Library

B.1 Tree Structure

```
type M5Node = {
  id : int ,
  splitAtt : int ,
  splitVal : real ,
  usedAtts : bool list ,
  classValues : real list ,
  dataValues : real list list
}
datatype errType = MAE| MSE| RelMSE
datatype M5Tree = empty | node of M5Node * M5Tree * M5Tree

type initType = {
                                 m_pruningMultiplier: real ,
                                 m_devFraction : real ,
                                 debug : int
}
```

B.2 Splitting Nodes

```
*
PART II - BUILD M5 MODEL TREE
*)
(*
II.1 - Helper function for Splitting tree
 + UsedAtts : find attributes tested in trees below for constructing
 linear regression model
 + separateList : separate a list to two sub-lists
 + sortByList : sort values of class according to values of a specific
   attribute. This attribute is also sorted
*)
(* UsedAtts : find attributes tested in trees below for constructing
linear regression model *)
fun reverseList (l) =
 let
  fun reverseHelp (l1,l2) =
  case l1 of
   nil => l2
```

```
    |h::t => reverseHelp(t,h::l2)
  in
   reverseHelp(l,nil)
  end

fun initUsedAtts (length, truePos) =
  let
   val posFromEnd = length + 1 - truePos
   fun initHelp (0,_,l) = l
   | initHelp (len,1,l) = initHelp (len-1,len, true::l)
   | initHelp (len,pos,l) = initHelp (len-1, pos-1,false::l)
  in
   initHelp(length, posFromEnd, nil)
  end

fun getUsedAtts (empty, numAtts) = initUsedAtts(numAtts, 0)
| getUsedAtts (node(crrNode, _,_), numAtts) = #usedAtts crrNode

exception UsedAtts_LENGTHNOTMATCH
fun UsedAtts(lTree,rTree, numAtts,splitAtt) =
let
 val initUsedAtts = initUsedAtts(numAtts, splitAtt)
 val leftUseAtts = getUsedAtts(lTree, numAtts)
 val rightUseAtts = getUsedAtts(rTree, numAtts)
 fun orLists (l1,l2,l3) =
  let
   fun orHelp (nil,nil,nil, l) = l
   | orHelp(nil,nil,h3::t3,_) = raise UsedAtts_LENGTHNOTMATCH
   | orHelp(nil,h2::t2,nil,_) = raise UsedAtts_LENGTHNOTMATCH
   | orHelp(nil,h2::t2,h3::t3,_) = raise UsedAtts_LENGTHNOTMATCH
   | orHelp(h1::t1,nil,h3::t3,_) = raise UsedAtts_LENGTHNOTMATCH
   | orHelp(h1::t1,h2::t2,nil,_) = raise UsedAtts_LENGTHNOTMATCH
   | orHelp(h1::t1,nil,nil,_) = raise UsedAtts_LENGTHNOTMATCH
   | orHelp (h1::t1, h2::t2,h3::t3,l) =
        orHelp(t1,t2,t3, (h1 orelse h2 orelse h3) :: l)
  in
   reverseList(orHelp(l1,l2,l3,nil)  )
  end
in
 orLists(initUsedAtts,leftUseAtts,rightUseAtts)
end

(*separateList : separate a list to two sub-lists*)
fun separaArrayHelp (source, des1, des2, pivot, counter) =
case counter = Array.length(source) of
 true => (des1, des2, pivot + 1)
 | false =>
  (
   case counter > pivot of
    false =>
        (case Array.update(des1,counter, Array.sub(source, counter)) of _ =>
                    separaArrayHelp (source, des1, des2, pivot, counter + 1))
    | true => (case Array.update(des2, counter - Array.length des1,
                    Array.sub(source, counter)) of _  =>
                 separaArrayHelp (source, des1, des2, pivot, counter + 1))
  )
fun separateArray(counter, leng, classVals : real array, attVals : real array)=
case counter = leng of
```

```
 true => (classVals, Array.fromList nil, counter)
 |false =>
   (
   case Real.compare(Array.sub(attVals, counter - 1),
        Array.sub(attVals, counter)) of
    EQUAL => separateArray(counter + 1, leng, classVals, attVals)
    | _ => separaArrayHelp (classVals, Array.array(counter,0.0),
                 Array.array(leng - counter,0.0), counter - 1, 0)

   )
(* sortByList : sort list of attribute Vals and class Vals
based on Vals of column ith *)

exception swap_OUTOFINDEX
fun swap (attVals, classVals, i, j) =
case i > Array.length attVals orelse j > Array.length attVals of
 true => raise swap_OUTOFINDEX
 | false =>
   let
    val attTemp = Array.sub(attVals, j )
        val classTemp = Array.sub(classVals, j )
   in
    Array.update(attVals,j, Array.sub(attVals, i));
    Array.update(attVals, i, attTemp);
    Array.update(classVals, j, Array.sub(classVals, i));
    Array.update(classVals, i, classTemp)
   end

fun partition(attVals :real array, classVals : real array,
                                              left, right,pivot)=
case Int.compare(left, right) of
 GREATER => right
 | EQUAL =>
   (
    case Array.sub(attVals, right) > pivot of
     true => right - 1
         | false => right
        )
 | LESS =>
   case Array.sub(attVals, left) < pivot of
    true => partition (attVals, classVals, left + 1, right, pivot)
        | false =>
          (
           case Array.sub(attVals, right) > pivot of
            true => partition (attVals, classVals, left, right - 1, pivot)
                | false =>
                  (swap(attVals, classVals, left, right);
                  partition (attVals, classVals, left + 1, right - 1, pivot))
      )


fun quickSort ( attVals :real array, classVals : real array,
               left, right) =
case left < right of
 true =>
 (
  let
   val middle = Real.ceil(Real.fromInt (left + right) / 2.0) - 1
```

```
    val pivot = Array.sub(attVals, middle)
    val pivotIndex = partition(attVals, classVals, left, right, pivot)
    val _ = quickSort(attVals, classVals, left, pivotIndex)
    val _ = quickSort(attVals, classVals, pivotIndex + 1, right)
   in
    (attVals, classVals)
   end
   )
| false =>  (attVals, classVals)

(*********** Separate column-wise dataVals *****************)
fun firstColumn dataVals =
case dataVals of
 nil => (nil, nil)
 | row :: rows =>
   let
    val (firstCol, otherCols) = firstColumn(rows)
   in
    ((hd row) :: firstCol, (tl row) :: otherCols)
   end
(*
II.2 - Functions for splitting M5 Tree
 + findBestSplit : find best point to split for an attribute.
    Return index of split and min standard deviation
 + split : split the whole dataset reaching this node.
    Do the findBestSplit over all attributes
 + buildTree : build M5 Tree
*)

(* findBestSplit : find best point to split. Return index of split
 and max reduction *)
fun findBestSplit(attVals : real array, classVals: real array,
   numInstances : int, counter : int, splitIndex, minSD): int * real =
case numInstances - counter of
   0 => (splitIndex, minSD)
   | _ : int =>
    case counter of
     1 =>
          (case  separateArray(1, numInstances, classVals, attVals) of
              (subArray1, subArray2, new_counter) =>
       case (SD_Calculation subArray1) * real(new_counter) +
            (SD_Calculation subArray2) * real(numInstances - new_counter)
             of minSDTemp =>
           case numInstances - new_counter of
            0 => (1, 9999999.0)
            | _ => findBestSplit(attVals, classVals, numInstances,
                                        new_counter + 1, new_counter,minSDTemp))
         |_ =>
       (case  separateArray(counter, numInstances, classVals, attVals) of
              (subArray1, subArray2, new_counter) =>
            case numInstances - new_counter of
            0 => (splitIndex, minSD)
            | _ =>
          case (SD_Calculation subArray1) * real(new_counter) +
                   SD_Calculation subArray2) * real(numInstances - new_counter) of
                 minSDTemp => case minSDTemp < ( minSD - 0.000001) of
                  true => findBestSplit(attVals, classVals,numInstances,
                               new_counter + 1,new_counter,minSDTemp)
```

```
                      |false => findBestSplit(attVals, classVals,numInstances,
                             new_counter + 1,splitIndex,minSD) )

(* split : split the whole dataset reaching this node.
Do the findBestSplit over all attributes *)
fun split (dataVals, classVals, numAtts, numInstances, counter,
 splitAtt, splitVal, minSD) : int*real*real =
case numAtts + 1 - counter of
 0 => (splitAtt,splitVal, minSD)
 | _ =>
   case counter of
         1 =>
          (
          case firstColumn dataVals of (attColumn, otherAtts) =>
          case quickSort( Array.fromList attColumn, Array.fromList classVals,
          0, numInstances - 1) of (sortedByArray, sortedClassVals) =>
      case findBestSplit( sortedByArray,sortedClassVals, numInstances,1,
                               0, 0.0)
               of (crrSplitIndex, crrMinSD) =>

      case Array.sub(sortedByArray, crrSplitIndex - 1) of crrSplitVal =>
       split(otherAtts,classVals,numAtts,numInstances,2,1,
                   crrSplitVal,crrMinSD))
         | _ =>
          case firstColumn dataVals of (attColumn, otherAtts) =>
      case quickSort( Array.fromList attColumn, Array.fromList classVals,
            0, numInstances - 1) of (sortedByArray, sortedClassVals) =>
      case findBestSplit( sortedByArray, sortedClassVals, numInstances,1,
            0, 0.0) of (crrSplitIndex, crrMinSD) =>

      case Array.sub(sortedByArray, crrSplitIndex - 1) of crrSplitVal =>
      case crrMinSD < ( minSD - 0.000001 ) of
           true =>
             (
             split(otherAtts,classVals,numAtts,numInstances, counter + 1,
                    counter, crrSplitVal ,crrMinSD))
           |false =>
                  split(otherAtts,classVals,numAtts, numInstances, counter + 1,
                    splitAtt, splitVal,minSD)
(* buildTree : build M5 Tree *)
exception separaData_LENGTH_DATA_CLASS_NOTMATCH
fun separaData (h::t : real list list,nil,_,_,_,_,_,_) =
   raise separaData_LENGTH_DATA_CLASS_NOTMATCH
          | separaData (nil,h ::t ,_,_,_,_,_,_) =
               raise separaData_LENGTH_DATA_CLASS_NOTMATCH
          | separaData (nil,nil,l_data,r_data,l_class,r_class,_,_) =
               (l_data,r_data,l_class,r_class)
          | separaData (h1::t1 : real list list, h2::t2,l_data,
               r_data,l_class,r_class,mSplitAtt,mSplitVal) =
                       if get_ith_element_of_list(h1,mSplitAtt) > mSplitVal
                       then separaData(t1,t2,l_data, h1::r_data,
                                      l_class, h2::r_class ,mSplitAtt,mSplitVal)
                       else separaData(t1,t2,h1::l_data, r_data, h2::l_class,
                                      r_class,mSplitAtt,mSplitVal)

fun buildTree(dataVals, classVals, numAtts, nodeID, factor_GlobalDev)
     : M5Tree * int=
case SD_Calculation_forList classVals of
```

```
(stdDev, numInstances) =>
  case (numInstances < 4 orelse stdDev < factor_GlobalDev )of
         true => (case {id = nodeID,
                                    splitAtt = 0,
                                    splitVal = 0.0,
                                    usedAtts = initUsedAtts(numAtts,0),
                            dataValues = dataVals,
                                    classValues = classVals} : M5Node of
                                     crrNode => (node(crrNode,empty,empty), nodeID))
         |false => case split(dataVals, classVals, numAtts, numInstances,
                       1, 0, 0.0,0.0) of(m_splitAtt,m_splitVal, m_minSD) =>
          case Real.compare(m_minSD, 9999999.0) of
            EQUAL => (case {id = nodeID,
                                    splitAtt = 0,
                                    splitVal = 0.0,
                                    usedAtts = initUsedAtts(numAtts,0),
                            dataValues = dataVals,
                                    classValues = classVals} : M5Node of
                                     crrNode => (node(crrNode,empty,empty), nodeID))
            | _ => (
            case separaData(dataVals, classVals, nil,nil,nil,nil,m_splitAtt,
                    m_splitVal) of
                        (l_dataVals, r_dataVals, l_classVals, r_classVals) =>
            case buildTree(l_dataVals,l_classVals, numAtts, nodeID + 1,
                    factor_GlobalDev) of (l_Tree,l_ID) =>
            case buildTree(r_dataVals, r_classVals, numAtts, l_ID + 1,
                    factor_GlobalDev) of(r_Tree,r_ID) =>
            case {id = nodeID,
                       splitAtt = m_splitAtt,
                       splitVal = m_splitVal,
                       usedAtts = UsedAtts(l_Tree,r_Tree, numAtts,m_splitAtt),
                       dataValues = dataVals,
                       classValues = classVals} : M5Node of
                        crrNode => (node(crrNode,l_Tree,r_Tree), r_ID) )
```

B.3 Pruning Tree

```
(*
PART III : PRUNE M5 TREE
*)
(*
III.1 - Import C function and implement functions to convert list/array to
C array
*)
val C_createCArr = _import "createCArr" public: int * int-> real array;
val C_toCArr = _import "toCArr" public: real array * int -> real array;
val C_freeVec = _import "freeVec" public : real array -> unit;
val C_printMat = _import "printMat" public : real array * int * int -> unit;
val C_mulVV = _import "mulVV" : int * real array * real array -> real;
val C_mulMV = _import "mulMV" public : int*int * int * real array *
              real array * real array -> unit;
val C_mulMM = _import "mulMM" public : int * int * int * int * real array *
              real array * real array-> unit;
val C_mseCal = _import "mseCal" public : int * int * real array *
                real array * real array * real array * real array -> real;
val C_maeCal = _import "maeCal" public : int * int * real array * real array
```

```
                * real array * real array * real array -> real;
val C_solve = _import "solve" public: int * int * real array * real array
                -> real array;

val C_numNotZeroCoef = _import "numNotZeroCoef" public :
                                                real array * int -> int;
val C_getElement = _import "getElement" : real array * int -> real;
fun fromList2Vec ls =
        let val v = Array.fromList (ls)
                val l = Array.length (v)
        in      C_toCArr (v, l)
        end
(*fun fromLists (lss) = fromList2Vec( List.concat (lss) )*)
fun fromLists (lss) = Array.fromList( List.concat (lss) )
fun mseCal (nRows, nCols, dataVals, coefVals, classVals) =
let
 val preds = Array.array(nRows, 0.0)
 val diff = Array.array( nRows, 0.0)

 val mse = C_mseCal(nRows,nCols,dataVals,coefVals,classVals,preds,diff)

in
 mse
end

fun maeCal (nRows, nCols, dataVals, coefVals, classVals) =
let
 val preds = Array.array(nRows, 0.0)
 val diff = Array.array( nRows, 0.0)
 val mae = C_maeCal(nRows,nCols,dataVals,coefVals,classVals,preds,diff)
in
 mae
end


fun CArr2SMLArray (arr, len) =
let
 val smlAr_init = array(len, 0.0)
 fun add (cAr, index, smlAr) =
  case index < 0 of
   true => smlAr
   | false => (update(smlAr, index, C_getElement(cAr,index));
                  add(cAr, index - 1, smlAr))
in
 add(arr,len-1,smlAr_init)
end

(*
III.2 - Implement linear regression model and relative helper functions
*)
exception selectedAtts_LENGTHNOTMATCH
fun selectedAtts (usedAtts, dataVals) : real list list  =
let
 fun selectAtts (nil, nil, l) = l
 | selectAtts (h::t,nil,_) = raise selectedAtts_LENGTHNOTMATCH
 | selectAtts (nil, h::t, _) = raise selectedAtts_LENGTHNOTMATCH
 | selectAtts( h :: t, s:: st, l) = if h then selectAtts(t,st, s::l)
                                     else selectAtts(t,st,l)
```

```
in
 map (fn x => reverseList(1.0 :: selectAtts(usedAtts,x,nil) )) dataVals
end

fun linearModel (usedAtts,dataVals :real list list , classVals :real list) =
 let
  val selectedDataVals = selectedAtts(usedAtts, dataVals)
  val nRows = length_of_list(classVals)
  val nCols = length_of_list(get_ith_element_of_list(selectedDataVals,1))
  val selectedData2Mat = fromLists(selectedDataVals)
  val classVals2Vec= Array.fromList classVals
  val A = Array.array(nCols * nCols,0.0)
  val B = Array.array(nCols,0.0)
  val _ = C_mulMM(1,nCols,nRows,nCols,selectedData2Mat,selectedData2Mat, A)
  val _ = C_mulMV(1,nRows,nCols,selectedData2Mat,classVals2Vec, B)
  val N = nCols
  val NRHS = 1
  val _ = C_solve(N,NRHS, A, B)
  val BsmlAr = CArr2SMLArray(B,N)
 in
  (selectedData2Mat, BsmlAr)
 end



fun linearModel_forArray ( selectedDataVals : real array, classVals :
                           real array, nRows : int, nCols: int) =
let
  val A = Array.array(nCols * nCols,0.0)
  val B = Array.array(nCols,0.0)
  val _ = C_mulMM(1,nCols,nRows,nCols,selectedDataVals,selectedDataVals, A)
(*  val _ = C_printMat(A, nCols, nCols) *)
  val _ = C_mulMV(1,nRows,nCols,selectedDataVals,classVals, B)
 (*val _ = C_printMat(B, nCols, 1)*)
  val N = nCols
  val NRHS = 1
  val _ = C_solve(N,NRHS, A, B)
 (* val _ = C_printMat(B,N,NRHS) *)
  val BsmlAr = CArr2SMLArray(B,N)
in
 BsmlAr
end


fun countSelectedAtts l =
case l of
 nil => 0
 |h::t => case h of
                       true => 1 + countSelectedAtts t
                       | false => countSelectedAtts t

fun pruneFactor(numInstances, numParams, initVals : initType) =
case Real.compare(numInstances, numParams ) of
 EQUAL =>
  10.0
 | _ =>
  (numInstances + numParams * (#m_pruningMultiplier initVals)) /
   (numInstances - numParams)
```

```

fun getNumInstances(tree : M5Tree) =
case tree of
 empty => 0.0
|node(crrNode,_,_) => Real.fromInt(length_of_list(#classValues crrNode))

(* return numParams, error of model(linear/subtree), list of LM ,tree *)

fun numCoef (linear , n) =
 C_numNotZeroCoef(linear ,n)

fun coefs2FullLinearCoefs (usedAtts, coefs, numAtts) : real array =
let
 val fullLinearCoefs = array(numAtts + 1, 0.0)
 fun fillFullLinearCoefs (nil , _ , _ ) = ""
 | fillFullLinearCoefs (false :: t, index1, index2) = fillFullLinearCoefs
                                                       (t, index1, index2 + 1)
 | fillFullLinearCoefs (true :: t, index1, index2) = (update(fullLinearCoefs ,
   index2, sub(coefs, index1));
   fillFullLinearCoefs (t, index1 + 1, index2 + 1))
 val _ = update(fullLinearCoefs, numAtts, sub(coefs, Array.length coefs - 1))
in
 fillFullLinearCoefs (usedAtts, 0, 0);
 fullLinearCoefs
end

fun getID(tree : M5Tree) =
case tree of
 empty => 0
|node(crrNode, _ , _ ) => #id crrNode

exception getUsedAtts_EMPTYTREE
fun getUsedAtts(tree : M5Tree) =
case tree of
 empty => raise getUsedAtts_EMPTYTREE
|node(crrNode, _ , _ ) => #usedAtts crrNode

fun getNumInstances (tree : M5Tree) =
case tree of
 empty => 0.0
 | node (crrNode, _ , _ )=> Real.fromInt(List.length ( #classValues crrNode))
(*
III.3 - Prune M5 tree
*)
(* update BoolList : update nth of TRUE elements to FALSE*)
exception updateBoolList_OUTOFBOUND
fun updateBoolList (ls : bool list, trueUpdate : int) : bool list =
let
 fun updateBoolList_helper ( nil, _ , _ ) = raise updateBoolList_OUTOFBOUND
 | updateBoolList_helper (true :: t, tU, tI) = if tU=tI then (false :: t)
                              else true :: updateBoolList_helper(t, tU, tI + 1)
 | updateBoolList_helper (false :: t, tU, tI ) =
                             false :: updateBoolList_helper(t, tU, tI)
in
 updateBoolList_helper (ls, trueUpdate, 0)
end

(* Greedy search for the best of linear model by
removing greedily coefficients in the model. Use Akaike criterion *)
fun removeOneCol (A : real array, nRows : int, nCols : int, colToRemove: int)
    : real array =
let
 val B = Array.array(nRows * (nCols - 1), 0.0)
 fun convert (indexA : int, indexB : int, indexColRemove: int) =
  case indexB = Array.length B of
   true => B
   |false =>
    (
     case indexA = indexColRemove of
     true => convert(indexA + 1, indexB, indexColRemove + nCols)
     |false => (Array.update(B, indexB, Array.sub(A, indexA));
                    convert(indexA + 1, indexB + 1, indexColRemove))
    )
in
 convert(0,0, colToRemove)
end

fun findBestLinear_inner
   (selectedDataVals : real array, selectedAtts : bool list,
   classVals : real array, index : int,
   (fullSE: real, initNumCoefs :int, numInstances :int),
   (m_coefs: real array, m_selectedDataVals : real array,
    m_selectedAtts : bool list, m_Akaike : real, improve :bool )) =
case countSelectedAtts selectedAtts + 1 of numCoefs =>
case numCoefs - 1 of crrNumCoefs =>
case index =  numCoefs - 1 of
 true => (m_coefs, m_selectedDataVals, m_selectedAtts, m_Akaike, improve)
 | false =>
 (
     case updateBoolList (selectedAtts, index)  of crrSelectedAtts =>
     case removeOneCol (selectedDataVals, numInstances, numCoefs, index)
           of crrSelectedDataVals =>
     case linearModel_forArray
          (crrSelectedDataVals, classVals, numInstances, crrNumCoefs)of linear =>
     case mseCal(numInstances, crrNumCoefs, crrSelectedDataVals,
                   linear, classVals) of crrMse =>
     case crrMse * crrMse * Real.fromInt numInstances of crrSE =>
     case (crrSE/fullSE) * Real.fromInt(numInstances - initNumCoefs) + 2.0 *
              (Real.fromInt crrNumCoefs) of crrAkaike =>
          case Real.compare(crrAkaike, m_Akaike) of
           LESS => (
             findBestLinear_inner(selectedDataVals, selectedAtts, classVals,
                             index + 1,
                             (fullSE, initNumCoefs, numInstances),
                        (linear,crrSelectedDataVals,crrSelectedAtts, crrAkaike, true))
           )
           | _ => findBestLinear_inner(selectedDataVals, selectedAtts, classVals,
                 index + 1,
             (fullSE, initNumCoefs, numInstances),
                 (m_coefs, m_selectedDataVals, m_selectedAtts, m_Akaike, improve))

)

fun findBestLinear(crrNode : M5Node) =
case List.length(#classValues crrNode) of numInstances =>
case countSelectedAtts (#usedAtts crrNode) + 1 of initNumCoefs =>
case Array.fromList (#classValues crrNode) of classValsArray =>
case linearModel (#usedAtts crrNode, #dataValues crrNode, #classValues
        crrNode) of (init_selectedDataVals, linear) =>
case mseCal(numInstances, initNumCoefs, init_selectedDataVals, linear,
        Array.fromList (#classValues crrNode)) of fullMSE =>
case fullMSE * fullMSE * Real.fromInt numInstances of fullSE =>
case Real.fromInt(numInstances - initNumCoefs) + 2.0 *
       (Real.fromInt initNumCoefs) of initAkaike =>
let
 fun findBestLinear_outer(coefs: real array, selectedDataVals : real array,
      selectedAtts: bool list, Akaike: real, improve : bool) =
 case findBestLinear_inner(selectedDataVals,selectedAtts, classValsArray, 0,
                          (fullSE, initNumCoefs, numInstances),
                          (coefs, selectedDataVals, selectedAtts, Akaike,improve)) of
      (m_coefs, m_selectedDataVals, m_selectedAtts, m_Akaike, m_improve) =>
 case m_improve of
  true => findBestLinear_outer(m_coefs, m_selectedDataVals, m_selectedAtts,
           m_Akaike, false)
  |false =>(m_coefs, m_selectedAtts, m_selectedDataVals)
in
 findBestLinear_outer (linear, init_selectedDataVals, #usedAtts crrNode,
          initAkaike, false)
end

(* Main pruning *)
exception prune_ONEBRANCH
fun prune( tree : M5Tree, paren_linearDict, numAtts : int, initVals)
: int * real * (int * int * real * real array) list * real array * M5Tree =
case tree of
 empty => raise prune_ONEBRANCH
 |node(crrNode,empty,empty) =>
  (

  case findBestLinear(crrNode) of (linear, selectedAtts, selectedDataVals) =>
  case coefs2FullLinearCoefs (selectedAtts, linear, numAtts) of
          fullLinearCoefs =>
  case length_of_list(#classValues crrNode) of m =>
  case mseCal(m, countSelectedAtts selectedAtts + 1,selectedDataVals,
                 linear, Array.fromList (#classValues crrNode)) of error =>
   (1,error,paren_linearDict, fullLinearCoefs,tree)
  )

 |node(crrNode, l_Tree, r_Tree) =>

  case prune(l_Tree, paren_linearDict, numAtts, initVals) of (l_params,
     l_error, l_dict, l_linear, l_prunedTree) =>
  case prune(r_Tree,l_dict,numAtts,initVals) of(r_params,r_error,r_dict,
     r_linear, r_prunedTree) =>

  case findBestLinear(crrNode) of
      (linear,selectedAtts,selectedDataVals)=>
  case coefs2FullLinearCoefs (selectedAtts, linear, numAtts)
      of fullLinearCoefs =>
  case length_of_list(#classValues crrNode) of m =>
  case mseCal(m,countSelectedAtts selectedAtts+1,selectedDataVals,linear,
         Array.fromList (#classValues crrNode)) of nodeError =>
  case nodeError * pruneFactor(Real.fromInt m, Real.fromInt
  (countSelectedAtts selectedAtts + 1),initVals) of adjustedNodeError =>
  case (l_error * (getNumInstances l_Tree) + r_error *
    (getNumInstances r_Tree)) / Real.fromInt m of treeError =>
  case treeError *  pruneFactor(Real.fromInt m,
    Real.fromInt (l_params + r_params + 1), initVals)
        of adjustedTreeError =>
  case adjustedNodeError > adjustedTreeError of
         true =>
           (l_params + r_params + 1, treeError, (getID l_prunedTree,
           #id crrNode,
           getNumInstances l_prunedTree, l_linear)::(getID r_prunedTree,
           #id crrNode, getNumInstances r_prunedTree, r_linear):: r_dict,
                       fullLinearCoefs, node(crrNode, l_prunedTree, r_prunedTree))
         |false =>
          (countSelectedAtts selectedAtts + 1,nodeError, r_dict,
                  fullLinearCoefs, node(crrNode, empty,empty))
```

B.4 Smoothing Tree

```
(*
PART IV : SMOOTHING
*)

datatype rconst = rconst of int * real *real
fun realLess( X : real, Y : real ) : bool = X < Y
fun realEqual( X : real, Y : real ) : bool = Real.compare( X, Y )=EQUAL
fun realAdd( X : real, Y : real ) : real = X + Y
fun realMultiply( X : real, Y : real ) : real = X * Y
fun realDivide( X : real, Y : real ) : real = X / Y
fun realSubtract( X : real, Y : real ) : real = X - Y
fun rconstLess ( ( X, C ) : real * rconst ) : bool =
 case C of rconst ( Compl , Stepsize, Current ) => realLess( X, Current)
fun tor (C : rconst) : real =
 case C of rconst(Compl, StepSize, Current) => Current

fun f( ( P, Q, N ) : real * real * real ) : real =
case tor( rconst (0, 5.0, 15.0) ) of K =>
 realDivide(
  realAdd(
   realMultiply(P, N),
   realMultiply(Q, K)
  ),
  realAdd(N, K)
  )
exception F_OUTOFBOUND
fun selectedF ( index, (P,Q,N)) =
case index of
 0 => f(P,Q,N)
 | _ => raise F_OUTOFBOUND

fun lookUpDict ( crrID : int, paren_linearDict :
    (int * int * real * real array) list) : int * real * real array =
case paren_linearDict of
 nil => (0, 0.0, Array.fromList nil)
 | (nodeID, parenID, numInstances, fullCoefs) :: t => if (nodeID =crrID)
   then (parenID, numInstances, fullCoefs) else lookUpDict (crrID, t)

```

```
fun smoothLeaf ( parenID, numInstances :real, leafLinearCoefs,
    paren_linearDict, fIndex) : string=
case parenID = 0 of
 true => ""
 | false =>
 (
  let
   val ( grandpaID , parenNumInstances , paren_linearCoefs ) =
        lookUpDict ( parenID, paren_linearDict)
   fun updateCoef ( index ) =
    case index = Array.length leafLinearCoefs of
        true => ""
        | false =>
      let
           val n = numInstances
           val p = Array.sub(leafLinearCoefs, index)
           val q = Array.sub(paren_linearCoefs, index)
           val temp = selectedF(fIndex, (p,q,n))
           val _ = Array.update(leafLinearCoefs, index, temp)

          in
      updateCoef ( index + 1)
          end
   in
    updateCoef(0);
    smoothLeaf ( grandpaID , parenNumInstances ,leafLinearCoefs,
                    paren_linearDict, fIndex)
   end
  )


exception smoothingProcess_EMPTYTREE
fun smoothingProcess(tree : M5Tree, paren_linearDict, fIndex) :string =
case tree of
 empty => raise smoothingProcess_EMPTYTREE
 | node (crrNode, empty, empty) =>
  (
   case lookUpDict (#id crrNode, paren_linearDict) of (parentID,
    numInstances, leafLinearCoefs) =>
    smoothLeaf (parentID, numInstances,leafLinearCoefs,paren_linearDict,
        fIndex))
 | node (crrNode, l_Tree, r_Tree) =>
  (
    smoothingProcess( l_Tree, paren_linearDict, fIndex);
        smoothingProcess (r_Tree, paren_linearDict, fIndex)
  )
```

B.5 Predicting new values

```
(*
PART V : PREDICT
*)
val testResults = ref ""

exception predictInstance_INSTANCELINEATNOTMATCH
exception predictInstance_NODENOTFOUND
exception predictInstance_INSTLENGTHNOTMATCH
fun predictInstance (instVals, tree, paren_linearDict :
   (int * int * real * real array) list) :real =
case tree of
 empty => raise predictInstance_NODENOTFOUND
 | node(crrNode, empty, empty) =>
         let
          val ( _ , _ , linear) = lookUpDict( #id crrNode, paren_linearDict)
          val instVals' = instVals  @ [1.0]
          val res= C_mulVV(length_of_list instVals',Array.fromList instVals',
          linear)

         in
          res
         end
 | node(crrNode, l_Tree, r_Tree) =>
     case get_ith_element_of_list
              (instVals, #splitAtt crrNode) - #splitVal crrNode > 0.0 of
          true => predictInstance (instVals, r_Tree, paren_linearDict)
          |false => predictInstance (instVals, l_Tree, paren_linearDict)

fun predictDataVals (dataVals, tree, paren_linearDict :
                                   (int * int * real * real array) list)=
case dataVals of
 nil => nil
 |h::t => predictInstance (h, tree,paren_linearDict) ::
                              predictDataVals(t, tree, paren_linearDict)

fun predict (dataVals, classVals, tree, paren_linearDict :
      (int * int * real * real array) list, typeErr : errType)
          : real list * real =
 case predictDataVals(dataVals, tree, paren_linearDict) of resVals =>
 case length_of_list classVals of numInstances =>
 case typeErr = MSE of
  true =>
   (case mseCal(numInstances, 1, Array.fromList resVals,
    Array.fromList[1.0], Array.fromList classVals ) of nodeError =>
        (resVals,    valOf(Real.fromString(Real.toString(nodeError)))))
  |false => (
   case typeErr = MAE of
    true =>(
       case maeCal( numInstances, 1, Array.fromList resVals,
                        Array.fromList [1.0],
                                    Array.fromList classVals ) of nodeError =>
                         resVals,valOf(Real.fromString(Real.toString(nodeError))))
              )
    |false => (
   case  SD_Calculation (Array.fromList classVals) of sdClassVals =>
   case mseCal(numInstances, 1, Array.fromList resVals,
      Array.fromList [1.0], Array.fromList classVals ) of nodeError =>
   (resVals,
    valOf(Real.fromString(Real.toString(nodeError /sdClassVals))))
     )
  )
```

B.6 Data Operations

```
*
PART VI : RUN THE MODEL
*)

(* VI.1 - Functions to read data from file *)

fun getFirstCharList (cli : char list) =
 case cli of
  nil => nil
 |h::t => if h = #"," orelse h = #"␣"
          then nil
                  else h::getFirstCharList(t)

fun parseLine (line : char list) : real list =
case line of
 nil => nil
 | #"," :: tail => parseLine(tail)
 | #"␣" :: tail => parseLine(tail)
 | _ =>
   let
    val numStr = implode (getFirstCharList line)
    val num = valOf (Real.fromString(numStr))
   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) =>
   let
    val num = valOf(Real.fromString
          (implode(getFirstCharList(explode(valOf( TextIO.inputLine fh))))))

   in
    num :: readLines (fh, nLines - 1)
   end
  | (true,false) => raise readClass_ERRORINPUT
in
 readLines(TextIO.openIn fileName, nLines)
end
(* VI.2 - Read training and validation data/class *)
```

```
fun write2File (str : string, file :string) =
let
 val outStr = TextIO.openOut file
 val _ = TextIO.output(outStr, str)
 val _ = TextIO.closeOut outStr
in
 ""
end

fun checkContain (ls : int list, value : int) : bool =
case ls of
 nil => false
 | h :: tl => if h = value then true else checkContain (tl, value)

fun rdomList (range : int, num: int, seed : Random.rand, resList)
        : int list =
case num = 0 of
 true => resList
 | 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 oneCVSplit (test_ratio: real) (X : real list list, y : real list) :
    (real list list * real list) * (real list list * real 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)
 val validRowsList = rdomList(numData - 1, numValid, seed, [])
 val (X_train, y_train, X_valid, y_valid) = trainValidSplit(X, y,
      validRowsList, 0, [], [], [], [])
in
 ((X_train, y_train),(X_valid, y_valid))
end

exception concatenateXy_LENGTHNOTMATCH
fun concatenateXy( nil, nil) = nil
  | concatenateXy( X :: Xs,y :: ys) = (X @ [y]) :: concatenateXy (Xs,ys)
  | concatenateXy( _, _ ) =raise concatenateXy_LENGTHNOTMATCH
```

```

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 evaluateTree(X_train, y_train, X_valid, y_valid, initVals, fIndex)
                : real * real  =
let
 val factor_GlobalDev = #m_devFraction initVals *
     ( #1 (SD_Calculation_forList  y_train))
 val (tree, ID) = buildTree(X_train, y_train, List.length (hd X_train),
      1, factor_GlobalDev)
 val ( _ ,
       _ ,
       paren_linearDict' : (Int.int * Int.int * real * real array) list,
       fullLinearCoeffs : real array,
       prunedTree) =
        prune( tree , nil, List.length (hd X_train), initVals)

 val paren_linearDict =( 1,0, 0.0, fullLinearCoeffs)::paren_linearDict'
 val _ = smoothingProcess(prunedTree, paren_linearDict, fIndex)
 val ( _ , MSEerror ) =
     predict (X_valid, y_valid, prunedTree, paren_linearDict , MSE)
 val ( _ , RelMSEerror ) =
     predict (X_valid, y_valid, prunedTree, paren_linearDict , RelMSE)
in
 (MSEerror, RelMSEerror)
end

fun CV_Evaluate(X_train_list, y_train_list, X_valid_list, y_valid_list,
    sumMSE, sumRelError, initVals, numCV, fIndex) : real * real =
case y_train_list of
 nil => (sumMSE/ Real.fromInt(numCV), sumRelError/ Real.fromInt(numCV))
 | h :: tail =>
  let
   val (MSEerror, RelError) = evaluateTree( hd X_train_list,
    hd y_train_list , hd X_valid_list , hd y_valid_list , initVals , fIndex)
   val _ = print ("MSE, RelMSE : " ^ Real.toString(MSEerror) ^" , " ^
        Real.toString(RelError) ^"\n")
  in
   CV_Evaluate( tl X_train_list , tl y_train_list , tl X_valid_list ,
                tl y_valid_list , sumMSE + MSEerror,
                                 sumRelError + RelError ,
                             initVals ,numCV, fIndex)
  end


fun CV_error (testRatio , numCV, initVals , fIndex) (X,y) =
let
 val (X_train_list , y_train_list , X_valid_list , y_valid_list) =
        nCVSplit(X, y, testRatio , numCV)
 val (MSEerr, RelMSEerr) = CV_Evaluate(X_train_list , y_train_list ,
        X_valid_list , y_valid_list , 0.0, 0.0, initVals , numCV, fIndex)
in
 (MSEerr, RelMSEerr)
end
```

Appendix C

ADATE specification file for Smoothing experiment

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


fun f( ( P, Q, N ) : real * real * real ) : real =
case tor( rconst (0, 5.0, 15.0) ) of K =>
case realMultiply(K,N) of NewK =>
 realDivide(
  realAdd(
   P,
   realMultiply(Q, NewK)
  ),
  realAdd(1.0, NewK)
  )


%%


type main_domain =  M5Tree * (int * int * real * real array) list
type main_range = (int * int * real * real array) list

fun memDictHash(
         ( NodeID, ParentID, NumInstances, Coefs ) : int * int * real *
                 real array
         ) : Word64.word =
   list_hash( fn X => X, [
      Word64FromInt64 NodeID,
      Word64FromInt64 ParentID,
      realHash NumInstances,
      list_hash( realHash, array_to_list Coefs )
      ]
      )

fun main_range_hash Xs = list_hash( memDictHash, 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 lookUpDict_ada ( crrID : int, paren_linearDict : (int * int * real *
 real array) list) : int * real * real array =
case paren_linearDict of
 nil => (0, 0.0, Array.fromList nil)
 | (nodeID, parenID, numInstances, fullCoefs) :: t=> if (nodeID = crrID)
    then (parenID,numInstances, fullCoefs) else lookUpDict_ada(crrID, t)

fun smoothLeaf_ada ( parenID : int, numInstances :real,
 leafLinearCoefs : real array, paren_linearDict :
 (int * int * real * real array)list ) : int =
case parenID = 0 of
 true => 0
 | false =>
 (
  let
   val ( grandpaID , parenNumInstances , paren_linearCoefs ) =
   lookUpDict_ada ( parenID, paren_linearDict)
   fun updateCoef ( index ) =
    case index = Array.length leafLinearCoefs of
        true => ""
        | false =>
      let
           val n = numInstances
           val p = Array.sub(leafLinearCoefs, index)
           val q = Array.sub(paren_linearCoefs, index)
           val testNan =
             Real.==(p,0.0) andalso Real.==(q,0.0)
           val val2Update = if testNan = true then 0.0
                      else f(p/(2.0*p + 2.0*q),q/(2.0*p + 2.0*q),1.0/n)
           val _ = Array.update(leafLinearCoefs, index,
              val2Update* (2.0*p + 2.0*q))
          in
      updateCoef ( index + 1)
          end
   in
    updateCoef(0);
    smoothLeaf_ada ( grandpaID , parenNumInstances ,leafLinearCoefs,
        paren_linearDict)
   end
  )

fun smoothingProcess_ada(
      tree : M5Tree,
      paren_linearDict : (Int64.int * Int64.int * real * real array)list
      ) : Int64.int =
case tree of
 empty => 0
 | node (crrNode, empty, empty) =>
  (case lookUpDict_ada ( Int64.fromInt( #id crrNode ), paren_linearDict)
    of (parentID, numInstances, leafLinearCoefs) =>
    smoothLeaf_ada (parentID, numInstances, leafLinearCoefs,
        paren_linearDict))
 | node (crrNode, l_Tree, r_Tree) =>
  (
    smoothingProcess_ada( l_Tree, paren_linearDict);
    smoothingProcess_ada (r_Tree, paren_linearDict)
  )

fun  arrCp ( arr : real array, newArr : real array, index : int) : int=
case index - Int64.fromInt( Array.length arr ) of
 0 => 0
 | _ => (Array.update(newArr, Int64.toInt index,
       Array.sub(arr, Int64.toInt index)); arrCp(arr, newArr, index + 1))

fun dictCp (dictSrc :  (int * int * real * real array) list) :
  (int * int * real * real array) list =
case dictSrc of
 nil => nil
 | (I1, I2, R, arr) :: remains =>
  let
   val newArr = Array.array(Array.length arr, 0.0)
   val _ = arrCp(arr, newArr, 0)
  in
   (I1,I2,R,newArr) :: dictCp(remains)
  end

(* type main_domain =  M5Tree * (int * int * real * real array) list *)

fun conv(  Paren_linearDict, PrunedTree ) : main_domain =
  ( PrunedTree,
    map( fn( A, B, C, D ) => ( Int64.fromInt A, Int64.fromInt B, C, D ),
        Paren_linearDict ) )


fun conv'(  Paren_linearDict, PrunedTree )  =
  ( map( fn( A, B, C, D ) => ( Int64.fromInt A, Int64.fromInt B, C, D ),
  Paren_linearDict ),
    PrunedTree
    )

fun main((PrunedTree, Paren_linearDict) : M5Tree *
 (int * int * real * real array) list ) :
 (int * int * real * real array) list =
let
 val Paren_linearDict_ada = dictCp(Paren_linearDict)

 val temp = smoothingProcess_ada(PrunedTree, Paren_linearDict_ada)
in
 Paren_linearDict_ada
end


val initVals : initType = {
 m_pruningMultiplier = 1.0,
 m_devFraction = 0.05,
 debug = 0
}
```

```
fun readXy( (X_file ,y_file , numOfIns)) =
let
 val y = readClass (y_file ,numOfIns)
 val X = readData (X_file ,numOfIns)
in
 (X, y)
end

fun trainValidSeparate ( oneCV_allXy , dataTrain , dataValid) =
case oneCV_allXy of
 nil => (dataTrain , dataValid)
 | ( train , valid) :: remains => trainValidSeparate
          ( remains , train :: dataTrain , valid :: dataValid)



fun build_unSmoothTree (typeErr: errType) ((X_train , y_train))  =
let

 val factor_GlobalDev = #m_devFraction initVals *
  ( #1 (SD_Calculation_forList  y_train))
 val (tree , ID) = buildTree(X_train , y_train , List.length(hd X_train),1,
  factor_GlobalDev)
 val numAtts = List.length (hd X_train)
 val ( _ ,
       _ ,
       paren_linearDict : (Int.int * Int.int * real * real array) list ,
       fullLinearCoeffs : real array ,
       prunedTree) =
        prune( tree , nil , typeErr , numAtts , initVals)
 val paren_linearDict = ( 1, 0, 0.0, fullLinearCoeffs)::paren_linearDict
 val lenCoefs = List.length(hd X_train) + 1

in
 (paren_linearDict , prunedTree)
end


fun pairXy ( X : real list list list , y : real list list ,
  res : ( real list list * real list) list) :
   ( real list list * real list) list  =
  case y of
    nil => res
   | hy :: tly => ( hd X, hy ) :: pairXy( tl X, tly , res )

fun treeValidData_nCV (testRatio , numCV ) ((X_file ,y_file , numOfIns))=
let
 val y = readClass (y_file ,numOfIns)
 val X = readData (X_file ,numOfIns)
 val (X_train_list , y_train_list , X_valid_list , y_valid_list) =
  nCVSplit(X, y, testRatio , numCV)
 val Xy_train_list = pairXy ( X_train_list , y_train_list , [])
 val tree_parenDict_list = List.map (build_unSmoothTree MSE)
       Xy_train_list
in
 (tree_parenDict_list , X_valid_list , y_valid_list)
end
```

```

val dataset = [
("/local/M5/2DPlanes_X_40768_10.csv","/local/M5/2DPlanes_y_40768.csv",
    40768),
("/local/M5/abalone_X_4177_9.csv","/local/M5/abalone_y_4177.csv", 4177),
("/local/M5/add10_X_9792_10.csv","/local/M5/add10_y_9792.csv", 9792),
("/local/M5/ailerons_X_13750_40.csv","/local/M5/ailerons_y_13750.csv",
   13750),
("/local/M5/bank8FM_X_8192_8.csv","/local/M5/bank8FM_y_8192.csv", 8192),
("/local/M5/bank32nh_X_8192_32.csv","/local/M5/bank32nh_y_8192.csv",
  8192),
("/local/M5/california_house_X_20640_8.csv","/local/M5/
␣␣␣california_house_y_20640.csv", 20640),
("/local/M5/CASP_X_45730_9.csv","/local/M5/CASP_y_45730.csv", 45730),
("/local/M5/CBM_X_11934_17.csv","/local/M5/CBM_y_11934.csv", 11934),
("/local/M5/CombinedCyclePowerPlant_X_9568_4.csv","/local/M5/
␣␣CombinedCyclePowerPlant_y_9568.csv", 9568),
("/local/M5/cpu_small_X_8192_12.csv","/local/M5/cpu_small_y_8192.csv",
  8192),
("/local/M5/delta_ailerons_X_7129_5.csv","/local/M5/
␣␣delta_ailerons_y_7129.csv", 7129),
("/local/M5/delta_ail_X_7129_5.csv","/local/M5/delta_ail_y_7129.csv",
  7129),
("/local/M5/anacalt_X_4052_7.csv","/local/M5/anacalt_y_4052.csv", 4052),
("/local/M5/cpu_act_X_8192_21.csv","/local/M5/cpu_act_y_8192.csv",8192),
("/local/M5/delta_elv_X_9517_6.csv","/local/M5/delta_elv_y_9517.csv",
   9517),
("/local/M5/RelationNetwork_Directed_X_53413_22.csv","/local/M5/
␣␣RelationNetwork_Directed_y_53413.csv", 53413),
("/local/M5/elevators_X_16599_18.csv","/local/M5/
␣␣elevators_y_16599.csv", 16599),
("/local/M5/fried_delve_X_40768_10.csv","/local/M5/
␣␣fried_delve_y_40768.csv", 40768),
("/local/M5/housePrice8_X_22784_8.csv","/local/M5/
␣␣housePrice8_y_22784.csv", 22784),
("/local/M5/housePrice16_X_22784_16.csv","/local/M5/
␣␣housePrice16_y_22784.csv", 22784),
("/local/M5/hwang_X_13600_11.csv","/local/M5/hwang_y_13600.csv", 13600),
("/local/M5/kin8nm_X_8192_8.csv","/local/M5/kin8nm_y_8192.csv", 8192),
("/local/M5/mv_X_40768_10.csv","/local/M5/mv_y_40768.csv", 40768),
("/local/M5/parkinsons_updrs_X_5875_21.csv","/local/M5/
␣parkinsons_updrs_y_5875.csv", 5875),
("/local/M5/pol48_X_15000_48.csv","/local/M5/pol48_y_15000.csv", 15000),
("/local/M5/pole_X_14998_26.csv","/local/M5/pole_y_14998.csv", 14998),
("/local/M5/puma8NH_X_8192_8.csv","/local/M5/puma8NH_y_8192.csv", 8192),
("/local/M5/puma32H_X_8192_32.csv","/local/M5/puma32H_y_8192.csv",8192),
("/local/M5/winequality_X_4898_11.csv","/local/M5/
␣␣winequality_y_4898.csv", 4898)
]

fun init_eval (I : Int.int, allTrees, allXvalid, allYvalid,init_RelMSE) :
  real list =
 case I = List.length allTrees of
  true => init_RelMSE
  | false =>
   let
    val prunedTree = #2 (List.nth (allTrees, I))
    val linearDict' = #1 (List.nth (allTrees, I))
    val linearDict'' = dictCp(linearDict')
    val linearDict =
      map( fn( A, B, C, D ) => ( Int64.toInt A, Int64.toInt B, C, D ),
           linearDict'' )
    val _ = smoothingProcess(prunedTree, linearDict)
    val Xvalid = List.nth(allXvalid, I)
    val Yvalid = List.nth(allYvalid, I)
    val ( _ , error ) =
    predict(
      Xvalid,
      Yvalid,
      prunedTree,
      linearDict,
      RelMSE)
  in
    init_eval(I + 1, allTrees,allXvalid, allYvalid,init_RelMSE @[error])
  end


val tree_Xvalid_yvalid_list_list = List.map (treeValidData_nCV (0.2, 5))
  dataset
val allTrees = List.concat (List.map #1 tree_Xvalid_yvalid_list_list)
val allXvalid = List.concat (List.map #2 tree_Xvalid_yvalid_list_list)
val allYvalid = List.concat (List.map #3 tree_Xvalid_yvalid_list_list)
val initRelMSE = init_eval(0, map( conv',allTrees ),allXvalid,allYvalid,
  [])
val Inputs = map(conv, List.take(allTrees, 100))
val Test_inputs = map( conv, List.drop(allTrees, 100))

val Funs_to_use = [
  "false", "true",
  "realLess", "realAdd", "realSubtract", "realMultiply",
  "realDivide", "tanh",
  "tor", "rconstLess"
  ]

val Abstract_types = [ ]
val Reject_funs = []
fun restore_transform D = D
fun compile_transform D = D
val print_synted_program  = Print.print_dec'

val AllAtOnce = false


exception MaxSyntComplExn
val MaxSyntCompl = (
  case getCommandOption "--maxSyntacticComplexity" of
    NONE => 200.0
  | SOME S => case Real.fromString S of SOME N => N
  ) handle Ex => raise MaxSyntComplExn



val OnlyCountCalls = false
val TimeLimit : Int.int = 20000000
val max_time_limit = fn () => Word64.fromInt TimeLimit : Word64.word
val max_test_time_limit = fn () =>Word64.fromInt TimeLimit :Word64.word
val time_limit_base = fn () => real TimeLimit

fun max_syntactic_complexity() = MaxSyntCompl
fun min_syntactic_complexity() = 0.0
val Use_test_data_for_max_syntactic_complexity = false

val File_name_extension = ""
val Resolution = NONE
val StochasticMode = false

val Number_of_output_attributes : Int64.int = 4



fun to ( G : real ) : LargeInt.int =
 Real.toLargeInt IEEEReal.TO_NEAREST(G * 1.0e14 )

structure Grade : GRADE =
struct
type grade = LargeInt.int
val NONE = LargeInt.maxInt
val zero = LargeInt.fromInt 0
val op+ = LargeInt.+
val comparisons = [ LargeInt.compare ]
fun toString( G : grade ) : string =
  Real.toString( Real.fromLargeInt G / 1.0E14 )
val N = LargeInt.fromInt 1000000 * LargeInt.fromInt 1000000
val significantComparisons = [ fn ( E1 , E2 )
  => LargeInt.compare ( E1 div N, E2 div N ) ]
fun toString ( G : grade ) : string =
  Real.toString ( Real.fromLargeInt G / 1.0E14 )
val pack = LargeInt.toString
fun unpack( S : string ) : grade =
  case LargeInt.fromString S of SOME G => G

val post_process = fn X => X
val toRealOpt = NONE
end

val Threshold =
  case getCommandOption "--threshold" of
    SOME S => case Real.fromString S of SOME N => N

val () = ( p"\n\nThreshold = "; print_real Threshold; p"\n\n" )

val N = List.length Inputs

fun output_eval_fun( exactlyOne( I : Int.int, _ ,
    paren_lineardict: main_range ) )
    : { numCorrect : Int.int, numWrong : Int.int, grade : Grade.grade }
        List.list = [
let
  val prunedTree = #2 (List.nth (allTrees, I))
  val Xvalid = List.nth(allXvalid, I)
  val Yvalid = List.nth(allYvalid, I)
  val ( _ , ActualError ) =
    predict(
      Xvalid,
      Yvalid,
      prunedTree,
      map( fn( A, B, C, D ) => ( Int64.toInt A, Int64.toInt B, C, D ),
          paren_lineardict ),
      RelMSE
      )
  val initError = List.nth( initRelMSE, I)
  val error =
    if I >= N then
      ActualError
    else if ActualError < ( 1.0 - Threshold ) * initError then
      ( 1.0 - Threshold ) * initError
    else
      ActualError
in
  if Real.==( error, 0.0 ) then
    { numCorrect = 100, numWrong = 0, grade = to error}
  else if error > 1.0E30 orelse not( Real.isNormal error ) then
   { numCorrect = 0, numWrong = 1, grade = to 1.0E30}
  else
    { numCorrect = 1, numWrong = 0, grade = to error}
end
]
```