MODEL TREE ANALYSIS WITH RANDOMLY GENERATED AND EVOLVED TREES

by

MARK MAKOTO SASAMOTO

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Information Systems,
Statistics, and Management Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2010

ABSTRACT

Tree structured modeling is a data mining technique used to recursively partition a data set into relatively homogeneous subgroups in order to make more accurate predictions on future observations. One of the earliest decision tree induction algorithms, CART (Classification and Regression Trees) (Breiman, Friedman, Olshen, and Stone 1984), had problems including greediness, split selection bias, and simplistic formation of classification and prediction rules in the terminal leaf nodes. Improvements are proposed in other algorithms including Bayesian CART (Chipman, George, and McCulloch 1998), Bayesian Treed Regression (Chipman, George, and McCulloch 2002), TARGET (Tree Analysis with Randomly Generated and Evolved Trees) (Fan and Gray 2005; Gray and Fan 2008), and Treed Regression (Alexander and Grimshaw 2006).

TARGET, Bayesian CART, and Bayesian Treed Regression introduced stochastically driven search methods that explore the tree space in a non-greedy fashion. These methods enable the tree space to be searched with global optimality in mind, rather than following a series of locally optimal splits. Treed Regression and Bayesian Treed Regression feature the addition of models in the leaf nodes to predict and classify new observations instead of using the mean or weighted majority vote as in traditional regression and classification trees, respectively.

This dissertation proposes a new method called M-TARGET (Model Tree Analysis with Randomly Evolved and Generated Trees) which combines the stochastic nature of TARGET

with the enhancement of models in the leaf nodes to improve prediction and classification accuracy. Comparisons with Treed Regression and Bayesian Treed Regression using real data sets show favorable results with regard to *RMSE* and tree size, which suggests that M-TARGET is a viable approach to decision tree modeling.

LIST OF ABBREVIATIONS AND SYMBOLS

a,b	Predictor variable linear combination coefficients for oblique split rules
AIC	Akaike Information Criterion
BIC	Bayesian Information Criterion
BTR	Bayesian Treed Regression
C	Value of X_i occurring in the training set
$C(\lvert T\rvert)$	Number of binary decision trees with $\lvert T\rvert$ terminal nodes
D	Subset of the levels K in the training set
$D(t)$	Deviance of node t
$D(T)$	Total deviance of a decision tree T
$D_{BIC}(T)$	*BIC* penalized deviance of a decision tree T
$E(X)$	Expected Value of random variable X
F	F statistic
$\hat{f}(\mathbf{x}_i)$	Leaf node regression model
$f_t(X;\theta_t)$	Linear regression function in the t-th terminal node
i	Node impurity
IQR	Inter quartile range
i(t)	Impurity of node t

K	Number of unique levels, k, for a categorical variable
K	Thousand
ln	Natural logarithm
LOG	Logarithm
$L(S)$	Log-likelihood of the training data for a saturated model
$L(T)$	Log-likelihood of the training data for the tree model T
M	Mean
mg	Milligrams
min	Minimum
MPa	Mega Pascals
n	Number of observations
n_t	Total number of observations in terminal node t
n_{kt}	Number of observations in terminal node t of class k
ng	Nanograms
p	P-value
p	Number of predictive variables
p_{kt}	Proportion of class k in node t
$p_{penalty}$	Number of effective parameters in the tree model
p_{split}	Probability of splitting a tree node
$P(\text{split} \mid d)$	Probability of splitting a tree node given depth d
$p(T)$	Prior distribution of tree sizes
$P(T \mid Y, X)$	Posterior probability limiting distribution

$q(T, T^*)$	Kernel which generates T^* from T^i
RMSE	Root Mean Squared Error
ROC	Receiver Operating Characteristic
SD	Standard deviation
SSE	Sum of Squared Errors
t	Terminal leaf node
T	Decision Tree
$\lvert T \rvert$	Number of terminal nodes in decision tree T
T^i	Candidate decision tree
TR	Treed Regression
$\mathbf{X}$	Predictor variable matrix (n x p) with elements x_{ij}
X_i	ith predictor variable
$\mathbf{y}$	Response (dependent) variable vector (n x 1) with elements y_i
{ }	Set
α	Base probability of splitting a node
β	Rate at which the propensity to split is diminished with depth
Δ	Change
$\in$	Element of a set { }
$\sum$	Summation
θ_{t1}, θ_{t2}	Regression coefficients using the independent variable $X(l_t)$
μg	Micrograms
<	Less than

$>$	Greater than
$\leq$	Less than or equal to
$\geq$	Greater than or equal to
$=$	Equal to
$\int$	Integral
$\mid$	Given, or conditioned upon

ACKNOWLEDGEMENTS

I would like to thank my dissertation committee members, Dr. Gray, Dr. Conerly, Dr. Hardin, Dr. Giesen, and Dr. Albright for their time and assistance in completing this work. I owe my deepest gratitude to my committee chairman, Dr. Brian Gray for the many, many hours of advising and support he has given in helping me reach this goal. Without his guidance and constant encouragement, this dissertation would not have been possible.

Dr. Hardin and Dr. Alan Safer, who was my advisor in my Master's program at CSULB, laid a solid foundation in their Data Mining courses that led me to choose it as both a field of research and career path.

I would also like to thank Dr. Conerly and Dr. Giesen for their financial support through the assistantships I've held on campus over the length of my residence here at UA. The experience gained has prepared me for the workplace as a practicing statistician.

Finally, this dissertation is dedicated to my wife Jasmin, whose love and support provided me the strength and perserverance I needed to see this through to completion.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Data Mining

Data mining is the analysis of (often large) observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner (Hand, Mannila, and Smyth 2001, chapter 1). Data mining techniques enable knowledge to be extracted from data in the form of statistical models in order to see how variables relate to each other and to better understand the underlying phenomena among them. Examples of data mining applications include predicting whether or not a borrower will default on a loan, estimating the value of a home, creating profiles of enrolling student populations, or discovering which supermarket products are likely to be purchased together. Data mining methods include neural networks, decision trees, cluster analysis, market basket analysis, and regression analysis, among others.

The data to be analyzed are typically organized into two components. The independent predictor variable values compose a matrix, $\mathbf{X}$, with p predictor variables and n observations, where each row represents a different observation and each column corresponds to a different variable:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \dots & \dots & \dots & \dots \\ x_{n1} & x_{n2} & \dots & x_{np} \end{bmatrix} \tag{1.1.1}$$

The dependent variable values form a vector $\mathbf{y} = (y_1, y_2, ..., y_n)$ corresponding to the observations in the **X** matrix.

The observations are often divided into three subsets in the data mining process: (1) *training data* used to build models, (2) *validation data* used for fine tuning and adjustments to the models, and (3) *testing data* used to see how the final model performs on new, unseen data.

1.2 Tree-Based Models

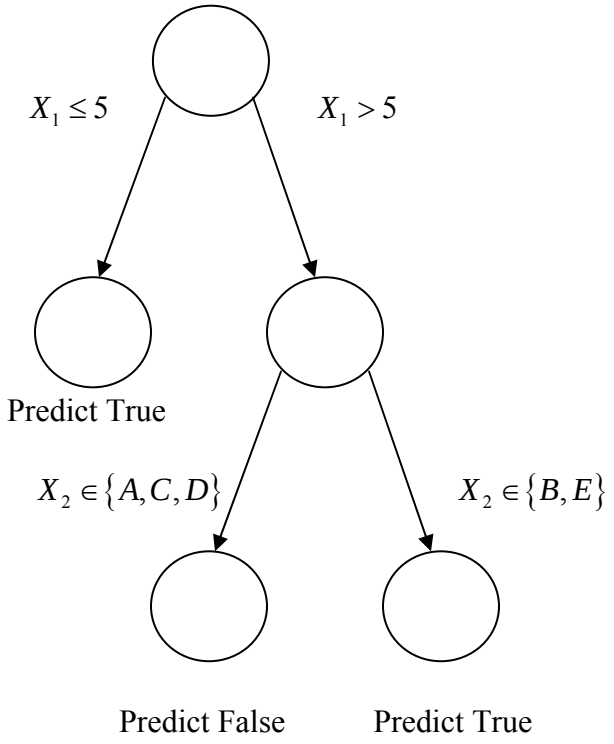
Tree-structured models are a form of supervised learning that use a set of continuous or categorical predictor variables to either classify a predetermined categorical target response variable (classification tree) or predict a continuous one (regression tree). Decision trees are created in the form of a top-down, flowchart-like model which recursively partitions a heterogeneous data set into smaller, more homogeneous sets where generalizations can be made to understand the interactions between predictors and how they relate to the target variable. Each partition of the data is called a "node," with the top node called the "root" and the terminal nodes called "leaves." The structure of the tree model itself gives insight and helps explain the underlying relationships in the data.

In algorithms following the traditional top-down induction of decision trees (TDIDT) format, such as CART (Classification And Regression Trees) (Breiman, Friedman, Olshen, and Stone 1984), CHAID (Chi-Square Automatic Interaction Detector) (Kass 1980), ID3 (Iterative Dichotomiser 3) (Quinlan 1986a), and C4.5 (Quinlan 1993), trees are constructed in a "greedy"

forward manner. Each split partitioning the data is made at a locally optimal location to improve a fitness measure, such as node impurity, sum of squared errors (*SSE*), or target variable variance. Each independent variable is considered for splitting and a univariate cutoff point is typically selected to divide the data so that the greatest fitness gain is found in the divided data. The tree induction process is computationally intensive, as each variable is searched across every level to find the locally optimal split rule. Split rules are not modified once determined and the data are partitioned accordingly. The split search algorithm is recursively applied to each newly formed node until a stopping criterion is met, such as too few observations to split or insignificant improvement in the fitness measure to justify the additional split. For trees predicting a categorical dependent variable, the classification assigned to leaf nodes is a weighted proportion of the observations partitioned into them. For a continuous dependent variable, the mean of the observations within a leaf node is given as the prediction for new data partitioned into that leaf node.

Allowing the tree to grow very large or continually splitting until pure leaf nodes are reached can lead to the problem of overfitting, which is the case when models are too specific to the training data used to create them and do not generalize well to new observations. Trees may be pruned back and fine-tuned using a hold-out data set, otherwise known as validation data, specifically withheld from the initial model building process to determine overall tree stability and nodes which may be removed to improve the generalization to future data. The final model is chosen to be the one that minimizes a misclassification or error function. Test data may then be used to assess the tree's performance on new data.

Figure 1.2.1. Example of a Decision Tree.



Decision trees offer many advantages over classical methods. The simpler graphical and more intuitive nature of the tree model makes it easier to interpret, without the need for a strong background in statistical methodology. Significant variables are clearly identified and a relative sense of importance conveyed by their positions in the tree. Mixed measurement scales are accommodated without any concessions in interpretability or visualization. No underlying assumptions about the linearity or normality of the data are necessary and decision trees are robust to outliers. Missing data can be handled without the need for imputation.

There are also several drawbacks to traditional decision tree models. The fitted model is not smooth, but rather a series of discontinuous flat surfaces forming an overall rough shape. Weighted polynomial smoothing techniques can be applied to smooth the disjoint transitions between leaf nodes (Chaudhuri, Huang, Loh, and Yao 1994). Trees can be very unstable and

sensitive to small perturbations in the training data, which makes them less reproducible due to the sequential nature of the model construction (Breiman 1996). Different samples of training data can produce tree models that are not similar. Minor changes in split rules at the top of the tree can have compounding impacts on the overall topology of the model.

The split selection process is biased towards selecting variables with many split points due to the greater possibility for significantly different partitions to be found (Loh and Shih 1997; Kim and Loh 2001, 2003; Hothorn, Hornik, and Zeileis 2006). Loh and Shih (1997), Loh (2002), and Kim and Loh (2001) propose alternative split search algorithms where the variable and split level selection are separated, effectively removing the selection bias.

The greedy approach to tree induction steers trees toward locations of local, rather than global optimality with each split (Murthy, Kasif, and Salzberg 1994; Brodley and Utgoff 1995; Fan and Gray 2005; Gray and Fan 2008). Not choosing the "correct" first split of the root node can lead the rest of the tree down a sub-optimal path. Though the focus of decision tree induction is local optimality at each splitting node because of computational constraints, the greedy algorithm does produce reasonably good trees (Murthy and Salzberg 1995a). Chan (1989), Norton (1989), and Hartmann, Varshney, Mehrota, and Gerberich (1982) propose tree induction algorithms incorporating limited "look-ahead" to search for the best combination of splits one to two more levels deeper before selecting a split rule as a compromise to the inability to exhaustively search the potential tree space. Murthy and Salzberg (1995b), and Elomaa and Malinen (2003) commented that limited look-ahead provides little benefit, if any at all, while adding increased computation time and size to the tree.

Genetic algorithms and Bayesian methods have been incorporated into the tree induction process as an alternative to search the tree space in a completely non-greedy fashion. Folino,

Pizzuti, and Spezzano (1999), Papagelis and Kalles (2000, 2001), Fan and Gray (2005), and Gray and Fan (2008) propose genetic algorithm-based methods of searching the tree space with overall optimality as a key criterion, not just one-step local optimality. Bayesian search methods have been proposed by Chipman, George, and McCulloch (1998), and Denison, Mallick, and Smith (1998), which incorporated Markov Chain Monte Carlo methods in the form of the Metropolis-Hastings algorithm to stochastically search the tree space.

Oblique splits consisting of linear combinations of predictor variables were introduced to make decision node splits more flexible and allow for finer interactions between variables to be represented in the splitting process. Breiman et al. (1984), Loh and Vanichsetakul (1988), Pagallo (1989), Utgoff and Brodley (1990), Bennett and Mangasarian (1992), Murthy et al. (1994, 1999), Brodley and Utgoff (1995), Chai, Huang, Zhuang, Yao, and Sklansky (1996), Cantu-Paz and Kamath (2003), Kretowski (2004), and Shali, Kangavari, and Bina (2007) propose methods for developing decision trees with oblique splits.

1.3 Genetic Algorithms

Originally formalized by Holland (1975), genetic algorithms provide a stochastically driven framework for finding near-optimal solutions for model parameters where exhaustive searches for globally optimal solutions are not computationally feasible. Their adaptation to decision tree induction enables trees to be induced with global optimality as a direct objective and eliminates the split search bias towards variables with a greater number of levels. Genetic algorithms efficiently search the tree space for near-optimal solutions and generally have better accuracy than the traditional TDIDT algorithms.

The algorithms are based on the principles of Darwinian natural selection, where a population of potential solutions to a problem is evolved through reproductive processes between its members, which compete for survival into future generations. Fitness functions evaluate the strength of each population member and determine its worthiness for survival.

Population chromosomes in the form of binary strings representing model parameters are typically randomly generated to form the first generation of potential parameter solutions in the genetic algorithm. Members are evaluated based on a fitness function and selected to reproduce either at random or proportionally to their fitness values, forming the next generation of potential solutions. Reproduction schemes take on three general forms: mutation, crossover, and cloning. In mutation, portions of the chromosomes are modified or replaced with different randomly selected parameter values. Crossover operations involve swapping substrings of parameter values between two chosen population members. The proportion of the population selected for reproduction and the fine details as to which members move on to the next generation will vary among algorithms. Other population members may be selected to survive unmodified in a process called cloning or elitism, while others may be randomly generated and inserted into the next iteration. This process of evolving generations continues until either the desired number of generations has been reached or a certain fitness-based stopping criterion is met. The final solution to the algorithm is the fittest model in the final generation.

Genetic algorithms differ from classical optimization and search methods in several key ways. Genetic algorithms operate and share information amongst a population of potential solutions, rather than make modifications on one solution at a time as in the Bayesian search methods (Chipman et al. 1998, 2002). Transitions are made from generation to generation probabilistically, as opposed to utilizing deterministic search elements. Population members are

selected with probability proportional to fitness in order to avoid narrowing down the search space too quickly by systematically selecting the strongest of the group and reducing the diversity needed for broader exploration of the tree space (Mitchell 1996).

1.4 Model Trees

Model trees feature the addition of functional models to the terminal nodes of the decision tree as an alternative to simply using the average as in regression trees or the majority rule as in classification trees. Model trees use the resulting regression residuals at each decision node in their splitting criteria and base the dependent variable predictions and classifications on the leaf node models. The presence of models in the leaf nodes enables the complexity of the tree to be shared between the tree structure and the regression models in the leaf nodes, making the trees smaller and more parsimonious. Karalič (1992), Quinlan (1992), Alexander and Grimshaw (1996), Torgo (1997a, 1997b), Wang and Witten (1997), Dobra and Gehrke (2002), Chan and Loh (2004), and Landwehr, Hall, and Frank (2005), and Vens and Blockeel (2006) proposed algorithms to induce trees with regression models in the leaf nodes to aid in prediction and to simplify the tree model structure.

1.5 The Focus of this Dissertation

This dissertation proposes a new decision tree induction algorithm, M-TARGET (Model Tree Analysis with Randomly Generated and Evolved Trees), which features the addition of models built into the leaf nodes of trees genetically induced. This research combines the elements of the TARGET (Fan and Gray 2005; Gray and Fan 2008) and model tree methodologies (Alexander and Grimshaw 1996; Chipman et al. 2002) to produce accurate

decision trees that are smaller due to the complexity of the model being divided between the tree structure and the leaf node model. The genetic algorithm framework removes the bias in variable selection and searches the tree space in an effective manner with the focus on global optimality, rather than local optimality.

CHAPTER 2

LITERATURE REVIEW

2.1 Top Down Induction of Decision Trees (TDIDT) Methodology

In algorithms following the traditional top-down induction of decision trees (TDIDT) format, such as CART, CHAID, ID3, and C4.5, the data are recursively partitioned into relatively homogeneous subsets, forming a tree structure. Predictions and classifications based on these leaf nodes are generally more accurate than those on the entire data set.

Beginning with the root node, traditional TDIDT based algorithms search over all possible split values to find the locally optimal location to partition the data in order to improve the chosen node-based fitness measure. Continuous or ordinal variables offer potential splits of the form $X_i \leq C$ versus $X_i > C$, where C is a value of X_i occurring in the training set. For ordered variables with K unique values, there are $K-1$ potential split points. Categorical variable splits are of the form $X_i \in D$ vs. $X_i \notin D$, where D is a subset of the levels from the training data. For categorical variables with K levels, there are $2^{K-1}-1$ potential split sets. To find the locally optimal split, each split value is used to temporarily partition the data and a fitness measure is used to evaluate the quality of this potential split. The best split rule is kept and the data is partitioned accordingly for further splitting and subdividing. The tree induction process is computationally intensive, as each variable is searched across every one of its unique levels to find the locally optimal split rule. Later advances in decision tree methodology have

streamlined the split search process and are more efficient because variable selection is conducted first, reducing the number of split points considered.

The split search algorithm is recursively applied to each child node until a stopping criterion is met, such as too few observations to split or insignificant improvement in the fitness measure to justify the additional split. Once fully built, trees may also be pruned back using the validation data, eliminating the nodes that reduce accuracy due to overfitting on the training data.

For trees predicting a categorical dependent variable, profit measures or loss functions can be used to weight the proportion of the observations partitioned into each leaf node. The classification assigned to a leaf node is the level with the greatest weight. For regression trees using a continuous dependent variable, the mean of the observations within a leaf node is given as the prediction for new data partitioned into that leaf node.

2.2 Other Selection Methods

QUEST (Quick, Unbiased, and Efficient Statistical Tree) (Loh and Shih 1997) separates the split variable and split value selection in order to remove the bias in selecting variables with a greater number of levels. In each node, the variable with the most significant ANOVA *F*-statistic is selected. A modified form of quadratic discriminant analysis is performed on the selected variable to find the split point used to further divide the data. Computation time is reduced because of fewer potential splits that need to be explored.

GUIDE (Generalized Unbiased Interaction Detection and Estimation) (Loh 2002) determines split rules in a fashion similar to QUEST. Continuous variables are divided into quartiles, while the levels of categorical variables are compared against whether the corresponding residuals are positive or non-negative. The independent variable with the most

significant chi-square statistic is selected for splitting. Since all levels of a categorical variable are included in the chi-square calculation, there is a bias toward selecting categorical variables in the splitting process. GUIDE features a bootstrap calibration for bias correction in which a scale factor is produced to adjust for the number of levels found in categorical variables in order to remove the bias in selecting them. Split value selection is based on either the exhaustive greedy search techniques found in CART or on the median as a split point. Interactions between independent variables are also directly searched over in the form of 2 x 4 contingency tables. Pairs of variables are split by their medians, or levels in the case of categorical variables, and by the signs of the model's residuals. Significant interactions are examined further to select one variable to split on. GUIDE's accuracy is favorable to CART on selected real data sets.

CRUISE (Classification Rule with Unbiased Interaction Selection and Estimation) (Kim and Loh 2001) uses the same split variable selection found in GUIDE. A Box-Cox transformation is performed on the selected split variable prior to a running a linear discriminant analysis in order to find the split point.

2.3 Oblique Split Selection Methods

Traditional decision tree induction utilizes univariate, axis-orthogonal split rules (e.g., $X_i \leq C$). Oblique splits consisting of linear combinations of predictor variables (e.g., $aX_i + bX_j \leq C$) were introduced to make decision node splits more flexible and allow for finer interactions between variables to be represented in the splitting process. Hyperplanes are produced that subdivide the data across more than one dimension at a time and allow more information to go into the splitting process. Algorithms may incorporate two or more split variables as an improvement to the axis orthogonal (or parallel) univariate split rules. As a

result, trees with oblique splits are shorter and more accurate than their univariate counterparts. In adding variables to the split search process, the computational complexity is greatly increased to find locally optimal splits via deterministic methods (Heath, Kasif, and Salzberg 1993; Murthy, Kasif, and Salzberg 1994; Murthy, Kasif, Salzberg, and Beigel 1999). There is also a trade-off in interpretability. Smaller trees are easier to interpret than larger ones, but decisions based on oblique splits within the tree structure can be more difficult to interpret (Utgoff and Brodley 1990; Murthy et al. 1994, 1999; Brodley and Utgoff 1995; Cantu-Paz and Kamath 2003).

Breiman et al. (1984) proposed a sequential split search algorithm as an addition to CART where variable coefficients are iteratively updated one at a time in an additive fashion. FRINGE (Pagallo 1989) proposed an iterative tree building algorithm which uses split variables chosen from previous trees to generate new variable combinations for future tree induction. PT2 (Perceptron Tree 2) (Utgoff and Brodley 1990) inputs each training observation sequentially into a bivariate linear threshold unit perceptron and updates split variables and values as it sees fit. The incremental handling of each training observation by the PT2 allows for future data to be incorporated into previously trained trees instead of requiring entirely new trees to be built. Bennett and Mangasarian (1992) introduced a linear programming algorithm to minimize a weighted average sum of the incorrectly classified training observations, which can be incorporated into the tree induction process. Brodley and Utgoff (1995) combine various deterministic methods of selecting split rules and utilize recursive least squares to learn coefficients and a hybrid of sequential forward selection (SFS) and sequential backward elimination (SBE) to select variables for the hyperplane. FACT (Fast Algorithm for Classification Trees) (Loh and Vanichsetakul 1988) and CRUISE suggest using a modified

version of linear discriminant analysis to form oblique splits. Variable and split value selection are separated in order to remove the bias in selecting variables with more levels. Principal components analysis is used on the variable correlation matrix to pre-select variables considered for splitting, where split values are chosen from linear discriminant functions calculated from the principal components whose eigenvalues are above a certain cutoff level.

Randomized stochastic techniques are introduced to curb the computational demands of finding optimal splits and avoid greedily splitting the tree space. The random nature of the stochastic search technique helps to avoid trapping induction algorithms in locally optimal regions (Heath et al. 1993; Murthy et al. 1994; Murthy et al. 1999). SADT (Simulated Annealing Decision Tree) (Heath et al. 1993) is a tree induction algorithm based on simulated annealing, which modifies randomly generated hyperplanes towards lower "energy" levels. Splits are determined iteratively by sequentially perturbing independent variable coefficients in the hyperplane and evaluated based on an entropy-based measure.

OC1 (Oblique Classifier 1) (Murthy et al. 1994; Murthy et al. 1999) combines the deterministic and stochastic approaches of previous methods by incorporating randomly generated restarts once locally optimal solutions are found deterministically. To escape locally optimal search spaces, solutions are randomly perturbed by random vectors or a new hyperplane can be generated to completely restart the search process. This method of generating oblique splits is similar to CART-LC (Breiman et al. 1984) which does not incorporate random restarts and only perturbs one independent variable coefficient at a time. OC1-ES (OC1-Evolution Strategies) (Cantu-Paz and Kamath 2003) extends OC1 by perturbing the coefficient vector with self-adaptive mutations. Over each generation, every coefficient is perturbed and kept if it

moves the split towards a better solution. OC1-ES performed comparably to OC1, though tree induction time was greatly reduced.

Genetic algorithms have been applied to search the space for optimal oblique splits. Genetic operations are carried out within each node where a single split is sought, as opposed to between the linear coefficients in different nodes. Trees are still induced in a top-down greedy fashion. BTGA (Binary Tree Genetic Algorithm) (Chai, Huang, Zhuang, Yao, and Sklansky 1996) applies genetic operations to strings of coefficients corresponding to the weights in the linear combination of independent variables comprising each oblique split. Split rules are initially generated randomly and then modified based on a weighted misclassification function. OC1-GA (OC1-Genetic Algorithm) (Cantu-Paz and Kamath 2003) performs similar crossover operations between hyperplane coefficients. Accuracy and tree size on several data sets is not as good as OC1, though run time is greatly reduced because splits are generated randomly, rather than deterministically. Kretowski (2004) uses an evolutionary technique involving pairs of training observations known as dipoles. The space is searched based on dipolar operators, where randomly selected dipoles are divided by adjusting coefficient weights and moving the hyperplane.

GIODeT (Genetically Induced Oblique Decision Tree) (Shali, Kangavari, and Bina 2007) offers a unique approach to genetically inducing oblique decision trees. Trees are induced in the familiar top-down greedy fashion with oblique hyperplanes. “Trees” of standard arithmetic operators are randomly induced and combined to form the expressions that generate the hyperplane. Crossovers and mutations are applied to the expression trees from generation to generation and evaluated based on the gain ratio. Results from several real world data sets were favorable compared to C4.5 in accuracy and tree size.

2.4 Genetic Algorithms in Decision Tree Induction

Complete exploration over the tree model space is a virtually impossible task to accomplish due to the number of potential trees that arise, even from a very limited data set (Chipman et al. 1998; Denison et al. 1998; Papagelis and Kalles 2001). Non-exhaustive search methods are needed to efficiently search the tree space. Traditional decision trees like CART (Breiman, et al. 1984) and variants using a "top-down" approach are induced by creating a sequence of locally optimal splits rather than a global optimization. This "greedy" forward selection process selects the predictor variable and value that optimally divide the data with respect to the target variable at each immediate step and permanently retains the split rule without considering later splits and the overall tree fitness. As a result, there is no guarantee that the tree will achieve global optimality (Murthy, Kasif, and Salzberg 1994; Brodley and Utgoff 1995; Fan and Gray 2005; Gray and Fan 2008).

A different form of split search is necessary in order to avoid the pitfalls of a "greedy" split selection process. In cases where finding optimal solutions is extremely difficult, the use of genetic algorithms is suggested as a way to efficiently search the space for a near-optimal solution (De Jong 1980; Hinton and Nowlan 1987; Oh, Kim, and Lee 2001). The genetic learning approach can be applied to decision tree construction as an alternative to the top-down induction process of "greedy" tree methods.

GA-ID3 (Bala, Huang, Vafaie, Dejong, and Wechsler 1995) and SET-Gen (Cherkauer and Shavlik 1996) apply genetic algorithms to select and limit the initial variables to use in decision tree model building prior to starting a traditional ID3 algorithm in cases where there are a large number of independent predictor variables. Genetic modifiers (mutation and crossover) were applied to the binary string representation of selected variables (0 = not included, 1 =

included) to be passed on to the tree induction process. Preselecting variables for the decision tree improved classification performance and reduced tree complexity when compared with traditional methods.

CGP (Cellular Genetic Programming) (Folino, Pizzuti, and Spezzano 1999) extended the genetic learning process from binary string operations to non-string decision tree structure objects to predict categorical dependent variables. The initial population of trees is grown randomly within "cells" and is free to mutate with other cell members through genetic operations on the tree nodes. Through each generation, every tree is subjected to a crossover, mutation, or reproduction operation. In a crossover, a subtree is swapped with the tree having the greatest fitness from the same cell. In a mutation, nodes are replaced with randomly generated subtrees. Reproduction operations make a copy of the tree to move on to the next generation. Periodically, a migration process is implemented where trees are moved between cells to distribute the genetic material. Findings by Zhu and Chipman (2006) illustrated that this "parallel universe" evolution strategy brings significant improvements over running a single evolutionary path. CGP did not produce trees that were as accurate, however the trees were much smaller and easier to interpret than those produced by C4.5.

GATree (Papagelis and Kalles 2000, 2001) is a similar algorithm to CGP, though it did not separate trees into cells. The initial population of trees is grown using randomly selected split variables and values installed in each decision node. This selection process removes the split selection bias towards variables with a greater number of splits because variable selection is now a separate step from split value selection. For categorical variables, one class is randomly selected to be the split set. For continuous variables, the split value is randomly selected from the range of observed values. Trees were selected for genetic operations inversely proportional

to their fitness. Better performing trees were less likely to be chosen for mutation or crossover, while weaker ones were more likely to be selected. Mutations consist of randomly changing the split value of the same split variable while crossover operations swap sub-trees between selected trees. Mutated trees replaced the weakest 25% of the population in each generation. On real data sets, the accuracy of the GATree algorithm was not significantly different than that of C4.5, though trees were much smaller in comparison. The drawback to this genetic approach however, is the increased computation time required to allow for the many generations of genetic operations needed to produce accurate trees.

TARGET improved on GATree by incorporating a more developed set of genetic operations, changing the fitness criterion, incorporating oblique splits into the decision nodes, and generalizing the framework to regression trees. The initial "forest" of trees is created by splitting nodes with probability, p_{split}, and installing randomly selected split variables and split rules into the decision nodes. Quantitative variables may be combined to form split rules in the form of linear combinations (oblique splits) as an option to improve over the univariate axis-orthogonal splits. Trees are selected for genetic operations directly proportional to their fitness levels and evolved to create subsequent generations. This is in contrast to GATree, which selects the lowest performing trees instead of the strongest to alter. In addition to the split value mutation found in GATree, TARGET adds three others: "split rule" mutation changes both the split variable and split value of a decision node; "node swap" mutation trades split rules between two nodes in the same tree; and "sub-tree swap" exchanges two randomly selected sub-trees within the same tree. TARGET uses a mix of crossovers, mutations, newly randomly generated trees, and retaining the best trees from the previous generation to form the next.

Though any tree fitness criterion can be used, TARGET uses the Bayesian Information Criterion (*BIC*) (Schwarz 1978) to evaluate the fitness of each tree. The *BIC* incorporates a stronger complexity penalty than the Akaike Information Criterion (*AIC*) (Akaike 1974), which would favor larger trees than the *BIC*.

TARGET performs very favorably compared to other methods involving one predictive tree, such as CART, QUEST, CRUISE, and Bayesian CART. Accuracy is not as good as ensemble methods unless oblique splits are incorporated as in TARGET-C2/C3, though interpretability is not lost as it is in ensemble voting procedures.

2.5 Bayesian Models

Bayesian methods were incorporated into decision trees as an alternative to the greedy approach by searching the tree space stochastically. These techniques combine prior and posterior probabilities to model the data and guide the tree building process. Other Bayesian based methods, such as those by Buntine (1992) and Nijssen (2008), use deterministic rather than stochastic searches.

Karalič and Cestnik (1991) incorporated a Bayesian approach into the pruning and classification stages of their own RETIS algorithm via the use of M-estimates. Prior probability distributions from the entire training data are weighted with the posterior probabilities from the subdivided data to make adjustments to the estimated dependent variable probability distributions in the tree nodes.

Bayesian CART (Chipman et al. 1998; Denison et al. 1998) incorporated Markov Chain Monte Carlo methods in the form of the Metropolis-Hastings algorithm to stochastically search the tree space. The Bayesian CART methods outperform CART because of the broader search

of the tree space and the non-greedy induction arising from the random generation of trees. The specification of the initial tree priors are carried out randomly, where tree nodes and their split rules are drawn in a similar fashion to the initial tree building process in TARGET. Nodes are split with probability p_{split} and split variables and values are chosen uniformly from the available predictors. The trees move from state to state and the posterior space is searched via changes similar to those found in genetic based algorithms, such as randomly growing, pruning, and changing split rules within the tree nodes. A key difference between the search methods of the Bayesian and genetic approaches is that the Bayesian method focuses on using a single tree to explore local modes by making small steps around the starting point and restarts itself to search across different areas, while the genetic algorithm implements a broad parallel search of the whole tree space with multiple trees simultaneously evolving.

A drawback to the Bayesian CART approach is that the tree space does not get as broadly searched as in TARGET due to the serial nature of its search technique. Results are largely dependent on the initial choice of prior as a starting point. In contrast, TARGET searches over a wider area simultaneously in a parallel nature and does not get stuck in searching over locally optimal regions. The individual trees within each forest in TARGET are able to share information in order to focus the search over more promising regions.

2.6 Leaf Models

To improve on the predictive accuracy of decision trees, model trees (Vens and Blockeel, 2005), otherwise known as hybrid tree learners (Torgo, 1997a) incorporate a functional model in each terminal leaf node as an alternative to simply using the average as in regression trees, or the majority rule as in classification trees. The greater predictive ability of the models in the nodes

reduces the need for complex tree structures and allows for fewer splits, resulting in shorter, simpler trees which are easier to interpret and understand (Quinlan 1992, Chaudhuri, Lo, Loh, and Yang 1995; Kim and Loh 2003, Vens and Blockeel 2005). The data modeling complexity can be shared between the tree structure itself and the model, as the tree models variable interactions through its recursive partitioning and piecewise linear functions.

Traditional methods of modeling, such as linear regression, perform well when the data take on a smoother, planar shape. By partitioning the complexity of the response surface and allowing the tree structure to handle variable interactions, the predictive power of linear models can be utilized locally on smoother subdivisions of the response surface while retaining overall parsimony of the decision tree (Chaudhuri, et al., 1995). Furthermore, hybrid trees are more stable when compared to CART, as they are more likely to evenly divide the data, rather than splintering off smaller groups at each split (Chaudhuri, Huang, Loh, and Yao, 1994). Models used in the growing of the tree also provide alternative methods for variable selection and stopping criteria.

Quinlan (1992) proposed replacing the naïve average as found in CART (Breiman, et. al, 1984) in regression tree terminal nodes with multiple regression models in the M5 algorithm, which was further clarified in M5' (Wang and Witten 1997). Trees are still grown greedily, in a similar fashion to CART, with the reduction of dependent variable standard deviation as the splitting criterion. The models built use only independent variables encountered in split rules in the leaf node's sub-tree and are simplified to reduce a multiplicative factor to inflate estimated error. Models are considered in non-leaf nodes during the pruning process, though the incoherence between the split selection and the leaf model selection still exists and is not fully

reconciled. M5 did produce shorter trees with greater accuracy than the regression trees produced by CART.

Torgo (1997a, 1997b) also replaces the average prediction in the leaf nodes as in M5, citing non-parametric kernel regression as the most advantageous method. On the chosen data sets, the incorporated kernel models used have the greatest improvements in predictive accuracy over the use of the average when compared to the other modeling approaches of *k*-nearest neighbors and linear regression. However, given that the tree was built using variance reduction as the splitting criterion, the results found are likely to be biased in favor of kernel regression and *k*-nearest neighbors methods because predictions are obtained using similar observations to a given query point. Had the objective predicting function of the terminal leaf nodes been taken into account during the splitting process, it would be possible to see different results. Trees split using reduction in variance about the mean are not able to find relationships as accurately because they do not take into account the regression models that are fit in the leaf nodes. Torgo (1997a) did not incorporate regression models into the split search process due to the additional computational complexity and increased runtime that it would involve.

SECRET (Scalable EM and Classification based Regression Tree) (Dobra and Gehrke 2002) also applied least squares regression models in the leaf nodes. Tree nodes are split in a two-stage process. Observations are clustered using the EM algorithm and split points are identified using quadratic discriminant analysis on those clusters. SECRET produced trees with comparable accuracy to GUIDE, but provided advantages in reduced computation time and greater scalability to larger data sets.

Treed Regression (Alexander and Grimshaw 1996), RETIS (Karalič 1992), and MAUVE ("M5' Adapted to use UniVariate Regression) (Vens and Blockeel 2006) utilize linear regression

models both in the creation of the tree structure as well as in the prediction process in their model building algorithms on continuous dependent variables. Similar to CART, these methods conduct exhaustive searches across all independent variables and their split points, fitting linear regressions on each potential left and right child node. In Treed Regression, simple linear regressions are carried out on the potential child nodes and the additional split, which reduces the overall tree's total leaf node *SSE,* is retained, while in RETIS, a weighted *MSE* metric resulting from multiple linear regressions and Bayesian analysis is similarly used. MAUVE uses multiple linear regression and seeks the split resulting in the greatest reduction in residual standard deviation within the child node models. In the case of Treed Regression, the additional requirements of searching over each of the potential regressors create much higher computational demands in having to repeatedly run regressions to find the one producing the smallest *SSE*. The multiple regression models built in local MAUVE would not create the same computational difficulties since it only needs to be run once per child node in each potential split consideration.

In a test run of twelve data sets, run times were much faster and trees were shorter on the whole for MAUVE than for Treed Regression, while not sacrificing much in the way of accuracy. Runs from RETIS were much longer than the others because of the incremental updating of the variance estimates in the nodes. Overall accuracy was largely data dependent, and each method performed comparably.

Treed Regression, RETIS, and MAUVE outperform CART and multiple linear regression on its own, although they still suffer from some of the same drawbacks. Trees are still developed without global optimality in mind and are built greedily, focusing on producing a sequence of locally optimal splits without any form of "look-ahead". Variable selection bias is still present,

where predictor variables with a greater number of potential split points have more opportunities in being selected over those with fewer.

Chan and Loh (2004); Landwehr, Hall, and Frank (2005); and Zeileis, Hothorn, and Hornik (2008) added to the work done previously in this area by generalizing the linear regression based framework to categorical variables via logistic regression models in the leaf nodes and proposing alternative variable selection algorithms, rather than exhaustive split searches in order to remove the bias in the variable selection process.

LOTUS (Logistic Tree with Unbiased Selection) (Chan and Loh 2004) separates the task of split variable and split point selection, removing the bias in choosing variables with more levels. Each continuous variable is separated into quintiles and a chi-square test of independence is performed between predictor and response variables. Categorical independent variables are tested in a similar fashion, with their levels forming the columns in the contingency table. Split level searches on the selected variable are restricted to the sample quantile points 0.3, 0.4, 0.5, 0.6, and 0.7 for continuous predictors. If a categorical split variable is selected, the levels are ranked based on the proportion of responses in the dependent variable and a cutoff point is selected to be the level that minimizes the sum of the dependent variable variances. The two levels above and below the selected cutoff are considered for the split level search. The split level minimizing the deviance from the simple or multiple logistic regressions fitted in the child nodes is selected and the data is partitioned accordingly. The logistic regression models in the leaf nodes classify the observations that fall within the respective nodes.

LMT (Logistic Model Tree) (Landwehr et al. 2005) expanded the C4.5 algorithm by fitting a simple logistic regression into every node. The logistic models in the subtrees are combined using the LogitBoost algorithm (Friedman, Hastie, and Tibshirani 2000) in a stepwise

manner. Combining new variables into previously built node models is computationally more efficient than building new ones at the leaf nodes.

In the model based recursive partitioning algorithm (MOB) proposed by Zeileis et al. (2008), models are created and the instability of each predictor variable is tested via heteroscedasticity estimators to determine whether there is a systematic pattern of variation that would justify further partitioning of the data. Only the variable exhibiting the greatest measure of stability, if significant, is considered for splitting. The split level that produces the best results in the child node objective functions is selected. Splitting stops when none of the predictor variables show significant levels of instability and the models in the terminal leaf nodes are used to make predictions for their respective observations. The depth of the tree can be controlled by adjusting the cutoff parameters on the instability functions, limiting the maximum depth of the tree, or adjusting the minimum observations required for a split search. MOB results on several data sets were favorable compared to CART, as the trees were generally more accurate and had fewer parameters.

The GUIDE algorithm for piecewise-constant models in the leaf nodes can be extended to fit linear regression models. This adjustment uses the residuals from a fitted simple or multiple linear regression model in a contingency table against each independent variable to determine the split rules in the same fashion as in the constant model case.

TSIR (Tree Structured Interpretable Regression) (Lubinsky 1994) and SMOTI (Stepwise Model Tree Induction) (Malerba, Esposito, Ceci, and Appice 2004) introduce the possibility of creating single-child regression nodes in addition to binary splits. These regression nodes use simple linear regressions to transform the dependent variable and remove the effects from independent variables. Predictions on new cases are made in the leaf nodes by adding the

coefficients from the regressions from the corresponding sub-trees. Global effects of independent variables can be identified in the regression nodes discovered higher in the tree structure, whereas the variable effect information taken from the terminal node regression models in other tree modeling algorithms is confined to the leaf nodes in which the models occur. TSIR outperformed CART in terms of accuracy in most cases, and trees produced were in most all cases shorter and more parsimonious. SMOTI outperforms RETIS in terms of accuracy due to the variable transformations and identification of global effects in the regression and performed and is similar to M5' in accuracy, while producing shorter and simpler trees. While TSIR and SMOTI trees may generally contain fewer nodes, are more accurate, and illustrate global effects, there is some loss of interpretability due to the complexity added in incorporating regression nodes within the non-terminal nodes of the tree structure. The regression nodes interrupt the flow of the partitioning in the tree structure and may be confusing.

Kim and Loh (2003) propose fitting a linear discriminant model in each node to serve as both a classifier on categorical data as well as possibly lending information to the split selection criteria. In this method, the best bivariate linear discriminant model is retained in each node to add predictive accuracy and reduce the tree size over the traditional CART methods. It is a general framework, and they suggest using it with any split selection method. Though discriminant analysis is bivariate in nature, univariate axis-orthogonal splits are still recommended.

Chaudhuri, et al. (1994, 1995) propose using the distribution of the residuals from the polynomial regression functions as the variable selection criterion in the tree. Linear models are built in nodes considered for splitting and the observations with positive and negative residuals are compared against each other in two sample t-tests as a form of a split search. The means and

variances of each independent variable are compared between both groups. The variable producing the most significant difference test is selected as the split variable, with the average of the two group means as the split point.

The residual based split selection process is shown to be much faster than CART's exhaustive search across all possible split points. Chaudhuri et al. (1994) further note that this approach has a tendency to select variables that are "more indicative of global trends," which shifts the focus on the overall fitness of the tree in the model building process. In looking at the averages of the two residual groups instead of individually observed variable values, this method is more robust to perturbations in the data, which under CART could potentially change the entire tree structure.

The Bayesian CART approach to tree induction was extended into model tree building in Bayesian Treed Regression models (Chipman et al. 2002). Trees are initially built using the same algorithm as in Bayesian CART, but with the addition of linear regression models added into the leaf nodes as an extension to the method. The leaf node models share data complexity with the tree structure and produce smaller and more accurate trees. Bayesian Treed models perform favorably to conventional trees, MARS (Multivariate Adaptive Regression Splines) (Friedman 1991), and neural networks using root mean squared error as a form of fitness criterion. It does suffer from the same drawbacks of being stuck in locally optimal areas and relatively confined to the initial choice of prior for a search space. Bayesian Treed Regression models depend upon a limited number of restarts to search other areas of the tree space.

CHAPTER 3

M-TARGET METHODOLOGY

In this chapter, four decision tree methodologies are described. The early developments of CART (Breiman, et al. 1984) and extensions to Treed Regression (Alexander and Grimshaw 1996) and Bayesian Treed Regression (Chipman et al. 2002) will be discussed. The focus of this dissertation is a new method, M-TARGET, which combines the genetically induced decision tree methodology of TARGET (Fan and Gray 2005) with regression models fitted to the data in the terminal nodes.

3.1 CART Methodology

CART is the leading example of a traditional top-down induction of decision trees (TDIDT) format, where the data are recursively partitioned into relatively homogeneous subsets, forming a tree structure. Predictions and classifications based on the terminal leaf nodes are generally more accurate than those based on the entire data set. Allowing the flexibility to divide the data into relatively homogeneous subgroups and create separate predictions for each terminal node gives tree models the potential to be more accurate than classical regression methods.

Beginning with the root node, CART based algorithms search over all possible split values to find the locally optimal location to partition the data in order to improve the chosen fitness measure. Every potential split value is used to temporarily partition the data and a node

impurity measure evaluates its quality on the newly divided groups. Continuous or ordinal variables offer potential splits of the form $X_i \le C$ versus $X_i > C$, where C is a value of X_i occurring in the training set. For ordered variables with K unique values, there are $K-1$ potential split points. Categorical variable splits are of the form $X_i \in D$ versus $X_i \notin D$, where D is a subset of the levels from the training data. For categorical variables with K levels, there are $2^{K-1}-1$ potential split sets. The tree induction process is relatively computationally intensive, as each variable is searched across every one of its unique levels to find the locally optimal split rule. Since split variable and split value selection are not separated, there is a bias in selecting variables with a greater number of potential split points over those that have fewer potential split points.

After all the potential splits are evaluated, the split rule that has the greatest reduction in dependent variable impurity is kept and the data is partitioned accordingly for further splitting and subdividing,

$$\Delta i = i\left(Parent\right) - \left\{i\left(LeftChild\right) + i\left(RightChild\right)\right\} \tag{3.1.1}$$

where i is the measure of impurity. In the classification tree context, the dependent variable impurity is measured based on the Gini index,

$$i\left(t\right) = 1 - \sum_{k=1}^{K} p_{kt}^{\,2} \tag{3.1.2}$$

where $i\left(t\right)$ is the impurity measure of node t, K is the number of dependent variable classes, and p_{kt} is the proportion of class k present in node t. Other variations on this measure exist, but the final tree selected is insensitive to the choice of specific splitting rule (Breiman et al. 1984). In the regression tree context, node impurity is measured by the variation of the dependent variable in the node.

The split search algorithm is recursively applied to each child node until a stopping criterion is met, such as too few observations to split or insignificant improvement in the node impurity measure to justify the additional split. Once fully built, trees may also be pruned back using the validation data, eliminating the nodes that reduce accuracy due to overfitting on the training data.

For trees predicting a categorical dependent variable, profit measures or loss functions can be used to weight the proportion of the observations partitioned into each leaf node. The classification assigned to a leaf node is the level with the greatest weight. For regression trees using a continuous dependent variable, the mean of the observations within a leaf node is given as the prediction for new data partitioned into that leaf node.

3.2 Treed Methodology

To improve on the predictive accuracy of CART decision trees, Alexander and Grimshaw's Treed Regression (1996) incorporates a functional model in each terminal leaf node as an alternative to simply using the average to make predictions in traditional regression tree models:

$$f_t\left(\mathbf{X};\theta_t\right) = \theta_{t1} + \theta_{t2} X\left(l_t\right), \tag{3.2.1}$$

where $f_t\left(\mathbf{X};\theta_t\right)$ is the linear regression function in the *t*-th terminal node with θ_{t1} and θ_{t2} as the regression coefficients using the independent variable $X\left(l_t\right)$ that minimizes the *SSE* given the data in the *t*-th terminal node. The greater predictive ability of the models in the nodes reduces the need for complex tree structures and allows for fewer splits, resulting in shorter, simpler trees, which are easier to interpret and understand.

In Treed Regression, the split search process is carried out similarly to CART, where all potential splits are considered. At each potential split partitioning, a simple linear regression is carried out in each child node for every independent variable. Instead of using variance reduction to measure the quality of a potential split, the pair of child node linear regression *SSE* values is used as the selection measure. The additional split which reduces the overall tree's total leaf node *SSE* is retained. Searching over each of the independent variables in the simple linear regression case creates much higher computational demands in having to repeatedly run regressions to find the one producing the smallest *SSE*.

Alexander and Grimshaw specified a minimum number of observations per node as a stopping criterion, though others such as the *BIC* may be substituted in as allowed by the algorithm's framework. Treed Regression may also be generalized to the multiple regression case.

Treed Regression outperforms CART, although it still suffers from some of the same drawbacks. Trees are still developed without global optimality in mind and are built greedily, focusing on producing a sequence of locally optimal splits without any form of "look-ahead". Variable selection bias is still present, where predictor variables with a greater number of potential split points have more opportunities in being selected over those with fewer.

3.3 Bayesian Treed Methodology

Bayesian Treed Regression (Chipman et al. 2002) incorporates leaf node linear models into the stochastic search algorithm of Bayesian CART as an alternative to the greedy approach of inducing tree structures, which removes the variable selection bias. The Bayesian decision tree induction algorithms search over candidate trees randomly generated and evolved from an

assumed prior distribution. Prior and posterior probabilities model the data and guide the tree building process.

An initial tree, T^0, is induced stochastically by randomly splitting terminal nodes beginning at the root, according to the two-parameter node splitting probability:

$$P(\text{split} \mid d) = \alpha(1+d)^{-\beta}, \tag{3.3.1}$$

where *d* is the depth of the node considered for splitting, with the root node having depth 0, and α, β are preselected split propensity parameters (.5 and 2, respectively, by default). If a node is split, a variable is randomly selected and one of its levels is randomly chosen to form the split rule. Splitting continues on all the subsequent child nodes until all are determined to be terminal.

Trees are evolved for a predetermined run length (default is 4,000 iterations) according to the Metropolis-Hastings search algorithm, which samples from a simulated Markov Chain of trees $T^0, T^1, T^2, \ldots$ Monte Carlo Markov Chain methods are used to stochastically search for high posterior probability trees using the limiting distribution $p(T \mid Y, X)$. Beginning with tree T^0, the transition from T^i to T^{i+1} is carried out by either keeping the current tree or replacing it with a candidate tree T^* with probability:

$$\alpha(T^i, T^*) = \min\left\{\frac{q(T^*, T^i)}{q(T^i, T^*)}\frac{p(Y \mid X, T^*)\,p(T^*)}{p(Y \mid X, T^i)\,p(T^i)}, 1\right\}, \tag{3.3.2}$$

where

$$p(Y \mid X, T) = \int p(Y \mid X, \Theta, T)\,p(\Theta \mid T)\,d\Theta, \tag{3.3.3}$$

$p(T)$ is the prior distribution of tree sizes, and $q(T, T^*)$ is the kernel which generates T^* from T^i by randomly choosing from four mutation steps. GROW randomly selects a terminal node in T^i and splits it into two child nodes using a randomly selected split rule. PRUNE randomly

selects the parent of a terminal node and removes both children from the tree. CHANGE randomly selects a splitting node and randomly reassigns its split rule. SWAP randomly selects two splitting nodes and exchanges their split rules.

The above process is repeated until the predetermined number of iterations has been reached. To diversify the search space, Chipman et al. suggest using "restarts" which repeats the whole algorithm to allow the search process to start over in a different region of the tree space. The process is repeated for the default number of 20 restarts, keeping the final tree from each run. The overall best tree from the group of finalists is selected as the champion tree.

3.4 M-TARGET Methodology

In this section, a new decision tree algorithm M-TARGET (Model Tree Analysis with Randomly Generated and Evolved Trees) is described that incorporates linear models in the leaf nodes of trees induced via genetic algorithms. This research extends the genetic algorithm elements of TARGET (Fan and Gray 2005, Gray and Fan 2008) with the incorporation of model tree methodologies to produce more accurate and smaller decision trees by shifting model complexity from the tree structure to the leaf node models. Similar to Bayesian CART and Bayesian Treed Regression, the TARGET genetic algorithm framework removes the bias in variable selection and searches the tree space in a more effective manner than TDIDT methodologies with the focus on global optimality, rather than local optimality.

The tree structure follows the familiar binary classification and regression tree format as found in the TDIDT methodology with the exception of prediction models replacing the prediction constants in the leaf nodes. Continuous or ordinal variables X_i have split rules of the form $X_i \le C$ versus $X_i > C$, where C is a value occurring in the training set. For an ordered

variable with *K* unique values, there are $K-1$ potential split points. Categorical variable splits are of the form $X_i \in D$ versus $X_i \notin D$, where *D* is a subset of the levels of the categorical variable. For a categorical variable with *K* unique levels, there are $2^{K-1}-1$ potential splits.

3.4.1 Forest Initialization

M-TARGET evolves a population of model trees referred to as the "forest" through a fixed number of generations. To begin the algorithm, a forest of a predetermined size is randomly generated and evaluated. To create each tree in the forest, the algorithm begins with a random split of the root node containing all of the training data. Splits are conducted by first randomly selecting a predictor variable, then randomly forming a split rule based on the values of that variable. If the selected split variable X_i is continuous or ordinal, one of the unique values in the training data, *C,* is randomly chosen to divide the observations in the node into two child nodes by forming a split rule of the form $X_i \le C$. In the case of a categorical variable X_i being selected, the levels moving to the left child node are randomly drawn from the set of all levels of X_i. The resulting split rule is of the form $X_i \in D$, where *D* is the set of chosen levels. In both cases, if the split rule condition is met, the observations are partitioned into the left child node, otherwise into the right child node. By separating the split variable and split value selections, the bias in split selection toward variables with more unique values is removed. This splitting process is recursively applied to the child nodes with the probability of a split:

$$p_{split} = P(\text{split} \mid d) = \alpha(1+d)^{-\beta} \qquad (3.4.1)$$

and $1-p_{split}$, the probability of becoming a terminal node, where α is the "base" probability of splitting a node, *d* is the depth of the node where the root has depth 0, and β is the rate at which

the propensity to split is diminished with depth (Chipman et al. 1998, 2002). Splitting continues until either all the nodes are terminal or a stopping criterion is reached, such as maximum depth or maximum tree size.

After the tree structure is created, the training observations are fed into the tree at the root node and recursively partitioned according to the split rules. Nodes are pruned back if there aren't enough observations in them to meet a minimum node size requirement, such as ten observations per node. Observations in the leaf nodes are then used to fit models for classifying or predicting the dependent variable in future observations. For continuous dependent variables, a simple or multiple linear regression model may be used or a logistic regression in the categorical dependent variable case.

This process of tree induction is repeated until the desired forest size is reached. The default value is 25 trees.

3.4.2 Tree Evaluation and Fitness Criteria

Each tree in the forest has a fitness value associated with it that is used in the evolution process. The M-TARGET framework allows for any overall model tree fitness criterion to be used in evaluating tree performance. Misclassification rate and residual sum of squares are good choices for classification and regression trees, respectively. Other options include the Bayesian information criterion (*BIC*) (Schwarz 1978) and Akaike information criterion (*AIC*) (Akaike 1974), which both feature a penalty for tree complexity. The *AIC* does not penalize the complexity brought on with additional model terms as severely as the *BIC* and has a tendency to overfit models (Koehler and Murphree 1988). Cavanaugh and Neath (1999) suggest that the *BIC* asymptotically identifies the true model with probability one, provided that the family of

candidate models under consideration includes the true model generating the data. For testing and evaluation, the *BIC* is selected because it favors smaller, more parsimonious models.

For classification trees, Gray and Fan (2008) give a *BIC* version of penalized deviance as

$$D_{BIC}(T) = D(T) + (K-1)|T| \ln n, \tag{3.4.2}$$

where $D(T)$ is the total deviance of the tree *T, K* is the number of classes in the dependent variable (restricted to $K = 2$ in the case of binary classification examples), *n* is the number of observations used to train the tree, $|T|$ is the number of terminal nodes in *T,* total deviance is defined to be the sum of the deviances of the terminal nodes in *T,*

$$D(T) = \sum_{t \in \{T\}} D(t), \tag{3.4.3}$$

where *t* is a terminal node in classification tree *T,* and node deviance is defined as

$$D(t) = 2 \ln L(S) - 2 \ln L(T) = 2 \sum_{k=1}^{K} n_{kt} \ln \left(\frac{n_t}{n_{kt}} \right), \tag{3.4.4}$$

where $L(S)$ is the log-likelihood of the training data for a saturated model, $L(T)$ is the log-likelihood of the training data for the tree model *T*, n_t is the total number of observations in terminal node *t* and n_{kt} is the number of observations in terminal node *t* of class *k*. The *BIC* penalty is in the $(K-1)|T| \ln n$ term of the right hand side, which penalizes for the complexity of the model. The $(K-1)|T|$ term accounts for the total number of parameters estimated in the terminal nodes of the model.

Gray and Fan (2008) suggest a modified form of the *BIC* penalized deviance for classification trees with the addition of a $2|T|-1$ term to account for an additional degree of freedom for each split rule determined for the tree:

$$D_{BIC}(T) = D(T) + \left[\left((K-1)|T|\right) + \left(2|T|-1\right)\right]\ln n = D(T) + \left[(K+1)|T|-1\right]\ln n\,. \quad (3.4.5)$$

The *BIC* for M-TARGET can be derived as

$$BIC = \max(\ln L) - \frac{1}{2}p_{penalty}\ln n = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln\left(\frac{SSE}{n}\right) - \frac{n}{2} - \frac{1}{2}p_{penalty}\ln(n), \quad (3.4.6)$$

where constant variance is assumed in the terminal nodes and $p_{penalty}$ is the effective number of parameters in the tree model, where $p_{penalty}$ equals 1 (for the constant variance term) plus the sum of the total number of regression coefficients in the leaf nodes and number of split rules in the tree structure. Additionally:

$$SSE = \sum_{t=1}^{T}\left\{\sum_{i=1}^{n_t}\left(y_{ti} - \hat{f}_t(\mathbf{x}_{ti})\right)^2\right\}, \quad (3.4.7)$$

where $\hat{f}_t(\mathbf{x}_i)$ is the regression model, n_t is the number of observations, and $(y_{ti}, \mathbf{x}_{ti})$ is the data in leaf node *t*, $i = 1, 2, ..., n_t$.

3.4.3 Forest Evolution

The forest is evolved across generations by applying genetic operations, including mutation, crossover, cloning, and transplanting. Each operation generates new trees from the current forest, which are inserted into the next generation and evaluated. Trees are selected proportional to their fitness level for genetic operations, so the overall forest moves towards regions of greater fitness in the model tree space.

There are seven different types of mutations currently featured in M-TARGET. Nodes are selected for mutation proportional to their node sizes. The default setting is to have 20 mutations carried out, randomly chosen from the seven possibilities. SPLIT SET mutation involves the random selection of a node and the replacement of the split values with a newly

randomly selected one. SPLIT RULE mutation replaces both the split variable and split value in a randomly chosen node. NODE SWAP mutation is a form of crossover which selects two nodes from the same tree and exchanges their split rules. GROW randomly splits a terminal node and selects a split rule at random. PRUNE randomly selects a terminal node and removes it along with its sibling node. CROSSOVER randomly selects a terminal node from two different trees and swaps their split rules. The better of the two newly created trees is inserted into the next forest generation. TRANSPLANT creates a new tree generated in the same manner as in the initial forest induction.

Figure 3.4.3.1. SPLIT SET Mutation

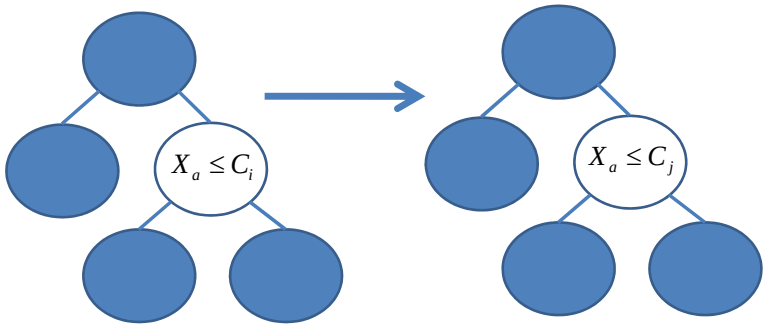


Figure 3.4.3.2. SPLIT RULE Mutation

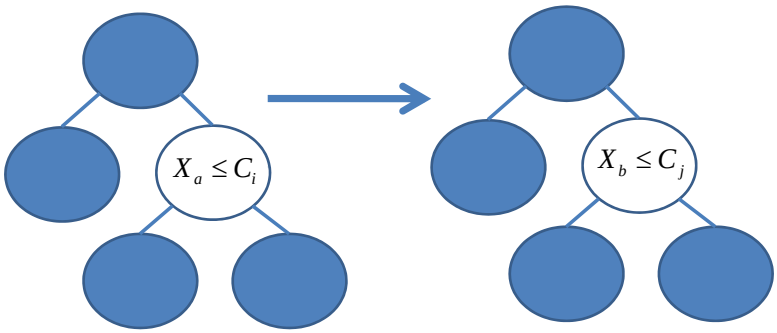


Figure 3.4.3.3. NODE SWAP Mutation

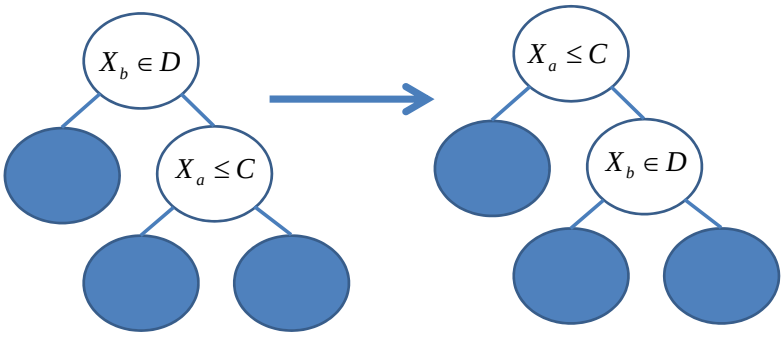


Figure 3.4.3.4. GROW Mutation

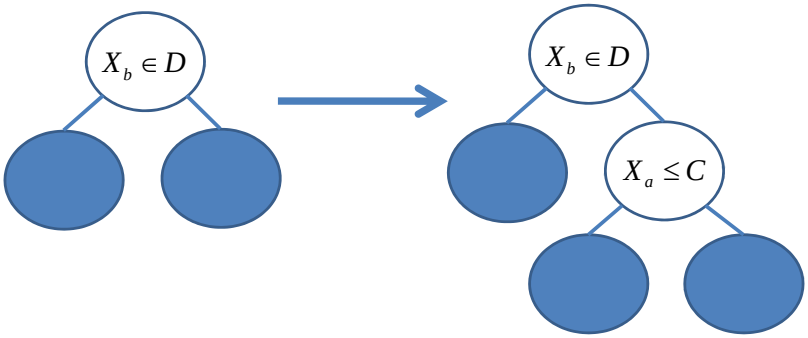


Figure 3.4.3.5. PRUNE Mutation

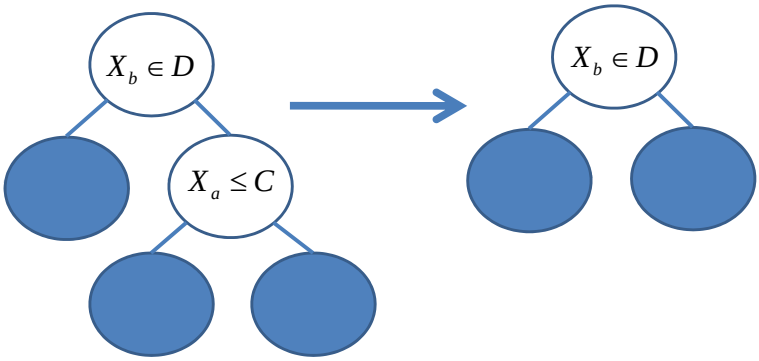
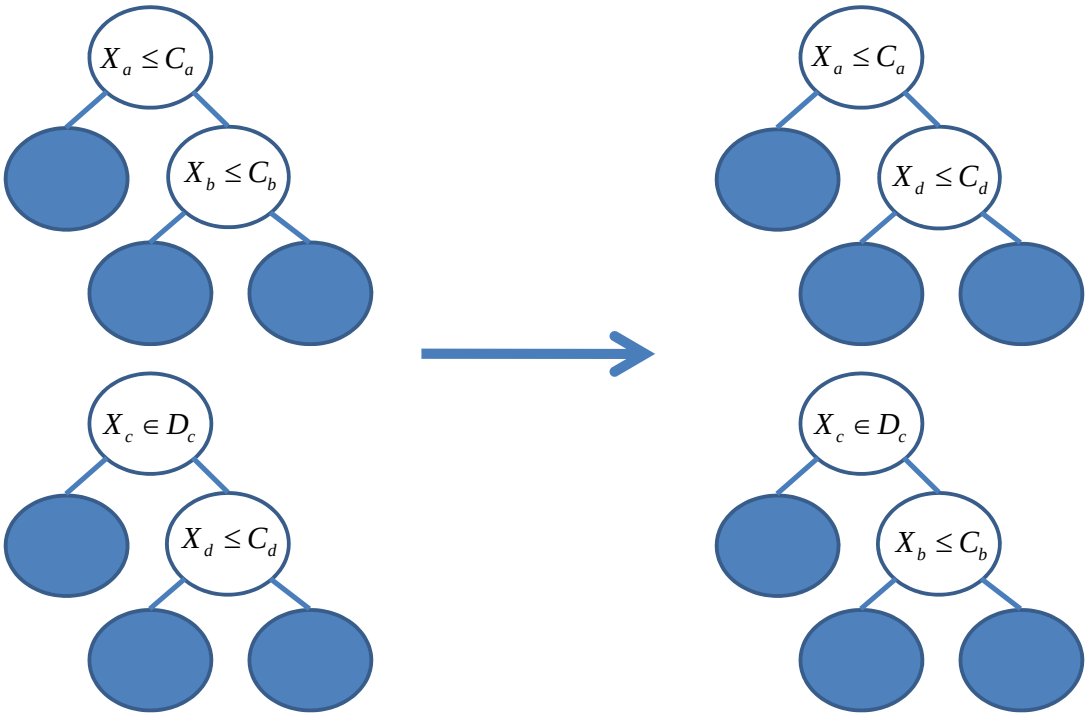


Figure 3.4.3.6. CROSSOVER Mutation



CLONE selects the best trees and inserts unmodified copies into the next generation. This guarantees that the forest's best performers will not be lost. By default, M-TARGET creates five clones per generation, creating a total of twenty-five trees per generation.

Once the forests are generated, the training observations are recursively partitioned according to the split rules and nodes are pruned back if empty. The best linear regression model is fit to the data in each leaf node and a model tree fitness measure evaluates the strength of the decision trees.

The evolutionary process continues until the predetermined number of generations has been reached. The final model selected at the end of the M-TARGET algorithm is the tree with the strongest fitness measure in the final forest. M-TARGET allows for restarts in a similar fashion to Bayesian Treed Regression where the evolutionary process is repeated, keeping the

best tree from each run. The overall best tree from the group of finalists is selected as the champion tree. Due to computational limitations, default settings are for 2,000 generation runs with 5 restarts.

CHAPTER 4

RESULTS AND COMPARISONS

In this section, five real data sets are used for performance comparisons among M-TARGET, Treed Regression (TR), and Bayesian Treed Regression (BTR). The data sets were chosen to represent a variety of situations, varying the number of categorical and continuous predictor variables as well as the number of observations. This should help expose strengths and flaws in each of the methods under different situations that commonly occur in real data.

10-fold cross-validation and repeated measures ANOVA are employed for comparisons among the methods. The same folds on each data set are used throughout the entire study for consistency. Four different settings of M-TARGET are tested and compared against the other two methods. The heavy computational requirements of the three methods in the R operating environment (R Development Core Team, 2010) prohibit more comprehensive runs to be made.

TR is a greedy algorithm which follows the traditional top-down induction of decision trees methodology. Comparisons with M-TARGET should illustrate practical differences between greedy and stochastically based induction algorithms. BTR is a stochastic non-greedy algorithm similar to M-TARGET, though BTR makes use of the Metropolis-Hastings algorithm to search the tree space one tree at a time, while M-TARGET uses a "forest" of 25 trees to simultaneously explore the tree space.

R 2.8.1 software is used to code and run M-TARGET and TR. BTR software is found in the tgp package in R (Gramacy 2007) and is considered to be the best implementation of the algorithm (Chipman 2009).

4.1 Real Data Sets Used for Comparison

4.1.1 Boston Housing

The Boston Housing data set is taken from the UCI Data Repository (Harrison and Rubinfeld 1978). There are 506 observations in the data set detailing housing values in suburbs of Boston. There are fourteen continuous variables included in the set:

1. CRIM: per capita crime rate by town
2. ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS: proportion of non-retail business acres per town
4. CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. NOX: nitric oxides concentration (parts per 10 million)
6. RM: average number of rooms per dwelling
7. AGE: proportion of owner-occupied units built prior to 1940
8. DIS: weighted distances to five Boston employment centres
9. RAD: index of accessibility to radial highways
10. TAX: full-value property-tax rate per $10,000
11. PTRATIO: pupil-teacher ratio by town
12. B: 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
13. LSTAT: % lower status of the population
14. LN(MEDV): natural log of the median value of owner-occupied homes in $1000's

This data set has been used frequently in the previous research discussed in the literature review.

4.1.2 Abalone

The Abalone data set is taken from the UCI Data Repository (Nash, Sellers, Talbot, Cawthorn, and Ford 1994). There are 4,177 observations in the data set, where the age of an abalone is predicted by its physical measurements. There are eight continuous variables and one categorical variable included in this data set:

1. SEX: gender (male, female, or infant)
2. LENGTH: longest shell measurement
3. DIAMETER: measurement perpendicular to length
4. HEIGHT: measurement with meat in the shell
5. WHOLE WEIGHT: whole weight of the abalone
6. SHUCKED WEIGHT: weight of the meat only
7. VISCERA WEIGHT: gut weight after bleeding
8. SHELL WEIGHT: weight after being dried
9. RINGS: count of the rings; this value +1.5 gives the age in years

RINGS is the dependent variable.

4.1.3 Concrete

The Concrete data set is taken from the UCI Data Repository (Yeh 2007). There are 1,030 observations in the data set measuring the compressive strength of concrete, which is a highly non-linear function of its age and its ingredients. There are nine continuous variables in

this data set, where the first seven are measured in kilograms per cubic meter. The data set is comprised of the following variables:

1. CEMENT
2. BLAST FURNACE SLAG
3. FLY ASH
4. WATER
5. SUPERPLASTICIZER
6. COARSE AGGREGATE
7. FINE AGGREGATE
8. AGE: measurement in days
9. CONCRETE: compressive strength measured in MPa

4.1.4 PM10

The PM10 data set (Aldrin 2004) is a subsample of 500 observations from a larger study where air pollution at a road is related to traffic volume and meteorological variables.

1. LOG(PM10 CONCENTRATION): hourly values of PM10 concentrations measured at Alnabru in Oslo, Norway between October 2001 and August 2003
2. LOG(CARS/HOUR)
3. TEMPERATURE: measured in degrees Celsius 2 meters above ground
4. WIND SPEED: measured in meters per second
5. TEMP DIFF: temperature difference between 25 and 2 meters above ground
6. WIND DIRECTION: degrees between 0 and 360
7. HOUR OF DAY

8. DAY NUMBER: number of days since October 1, 2001

This data set has four variables with a periodic cyclical pattern. Decision trees have the flexibility to partition periodic variables into relatively linear subgroups and make better use of them than classical linear methods.

4.1.5 Plasma Beta

The Plasma Beta data set (Nierenberg, Stukel, Baron, Dain, and Greenberg 1989) investigates the relationship between personal characteristics and plasma concentrations of beta-carotene. There are 315 observations, 9 continuous predictor variables, and 3 categorical predictor variables.

1. AGE
2. SEX: gender (male or female)
3. SMOKSTAT: smoking status (never, former smoker, current smoker)
4. QUETELET: $\frac{weight}{height^2}$
5. VITUSE: vitamin use (yes, fairly often; yes, not often; no)
6. CALORIES: number of calories consumed per day
7. FAT: grams of fat consumed per day
8. FIBER: grams of fiber consumed per day
9. ALCOHOL: number of alcoholic drinks consumed per week
10. CHOLESTEROL: cholesterol consumed (mg/day)
11. BETADIET: dietary beta-carotene consumed (μg /day)
12. RETDIET: dietary retinol consumed (μg /day)
13. BETAPLASMA: plasma beta-carotene (ng/day)

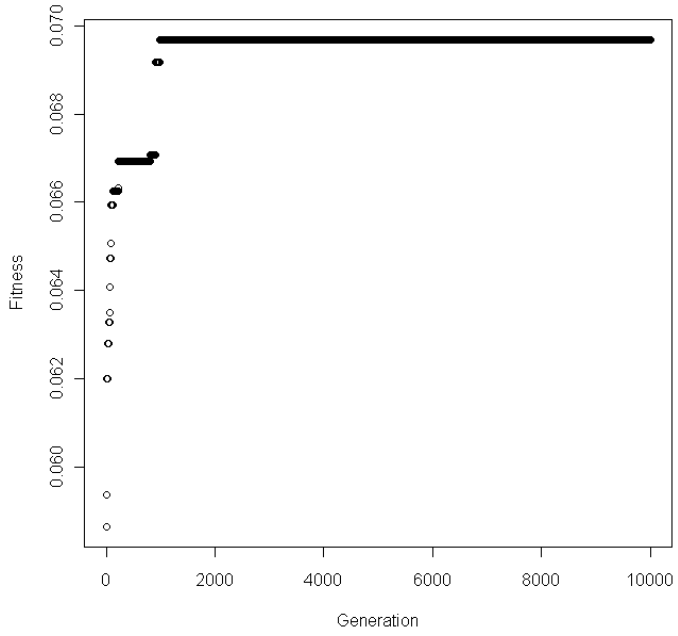
4.2 Comparison of Chosen M-TARGET Settings

Four different combinations of settings are chosen for trials in M-TARGET, varying the number of generations, restarts, and components of p_{split}.

4.2.1 Selection of the Number of Generations and Restarts

The levels of restarts and generations are chosen to be 4 restarts (for a total of 5 starts) of 2,000 generations or 1 restart (for a total of 2 starts) of 5,000 generations each, totaling 10,000 generations per run. Trials have shown that a minimum of 2,000 generations is typically needed for the evolutionary process to stabilize and 5,000 generations is a reasonable limit given the long computation time in the R environment. Figure 4.2.1 is a graph illustrating typical improvement in fitness measured by the inverse of the overall *SSE* during the evolutionary process for several data sets. Generally speaking, pilot studies suggest that improvements to the trees plateau once 2,000 to 3,000 generations have been reached.

Figure 4.2.1.1. Example Graph of Fitness Level Versus Generation



4.2.2 Selection of the Components of p_{split}

Two combinations of p_{split} components (α, β) are selected to be (0.4, 0) and (0.5, 2), respectively. The latter is selected by Chipman et al. (2002) as the default setting in BTR because it puts a heavy restriction on tree sizes with the intent that the modeling complexity be shifted to the leaf node regressions. The other chosen value of (0.4, 0) is selected from the simulation of 100,000 trees randomly generated. Levels of β considered included 0.5, 1, 1.5, and 2. Each of these levels is compared against varying values of α to determine the expected tree sizes for each combination of settings. The following tables resulted from the simulation runs of 100,000 randomly generated trees, without having data partitioned and trees pruned accordingly.

Table 4.2.2.1. Total Nodes in Randomly Generated Trees, $\beta = 0$

Alpha	Mean	SD	25th Pct.	Median	75th Pct.	IQR
0.10	3.497	1.180	3	3	3	0
0.15	3.845	1.700	3	3	5	2
0.20	4.327	2.422	3	3	5	2
0.25	4.992	3.450	3	3	5	2
0.30	6.014	5.126	3	5	7	4
0.35	7.610	8.077	3	5	9	6
0.40	11.032	15.330	3	5	11	8
0.45	21.075	44.459	3	7	19	16

Table 4.2.2.2. Total Nodes in Randomly Generated Trees, $\beta = 0.5$

Alpha	Mean	SD	25th Pct.	Median	75th Pct.	IQR
0.10	3.317	0.865	3	3	3	0
0.15	3.508	1.140	3	3	3	0
0.20	3.727	1.434	3	3	5	2
0.25	3.980	1.730	3	3	5	2
0.30	4.261	2.072	3	3	5	2
0.35	4.572	2.435	3	3	5	2
0.40	4.978	4.859	3	3	5	2
0.45	5.412	3.334	3	5	7	4
0.50	5.949	3.888	3	5	7	4
0.55	6.544	4.485	3	5	9	6
0.60	7.267	5.203	3	5	9	6
0.65	8.129	6.036	3	7	11	8
0.70	9.235	7.081	3	7	13	10
0.75	10.451	8.157	5	7	13	8
0.80	12.015	9.563	5	9	17	12
0.85	13.897	11.205	5	11	19	14
0.90	16.293	13.217	7	13	23	16
0.95	19.154	15.404	7	15	27	20

Table 4.2.2.3. Total Nodes in Randomly Generated Trees, $\beta = 1$

Alpha	Mean	SD	25th Pct.	Median	75th Pct.	IQR
0.10	3.220	0.690	3	3	3	0
0.15	3.334	0.870	3	3	3	0
0.20	3.463	1.050	3	3	3	0
0.25	3.595	1.205	3	3	3	0
0.30	3.738	1.380	3	3	5	2
0.35	3.901	1.560	3	3	5	2
0.40	4.063	1.720	3	3	5	2
0.45	4.239	1.882	3	3	5	2
0.50	4.444	2.088	3	3	5	2
0.55	4.640	2.258	3	3	5	2
0.60	4.865	2.469	3	5	5	2
0.65	5.112	2.677	3	5	7	4
0.70	5.369	2.887	3	5	7	4
0.75	5.624	3.102	3	5	7	4
0.80	5.966	3.353	3	5	7	4
0.85	6.241	3.570	3	5	7	4
0.90	6.617	3.828	3	5	9	6
0.95	6.968	4.068	3	5	9	6

Table 4.2.2.4. Total Nodes in Randomly Generated Trees, $\beta = 1.5$

Alpha	Mean	SD	25th Pct.	Median	75th Pct.	IQR
0.10	3.150	0.557	3	3	3	0
0.15	3.225	0.691	3	3	3	0
0.20	3.307	0.818	3	3	3	0
0.25	3.389	0.931	3	3	3	0
0.30	3.477	1.034	3	3	3	0
0.35	3.572	1.142	3	3	3	0
0.40	3.658	1.240	3	3	5	2
0.45	3.761	1.349	3	3	5	2
0.50	3.865	1.441	3	3	5	2
0.55	3.967	1.542	3	3	5	2
0.60	4.077	1.643	3	3	5	2
0.65	4.190	1.732	3	3	5	2
0.70	4.306	1.841	3	3	5	2
0.75	4.429	1.945	3	3	5	2
0.80	4.572	2.061	3	3	5	2
0.85	4.691	2.146	3	5	5	2
0.90	4.825	2.234	3	5	5	2
0.95	4.974	2.346	3	5	7	4

Table 4.2.2.5. Total Nodes in Randomly Generated Trees, $\beta = 2$

Alpha	Mean	SD	25th Pct.	Median	75th Pct.	IQR
0.10	3.102	0.455	3	3	3	0
0.15	3.152	0.559	3	3	3	0
0.20	3.208	0.656	3	3	3	0
0.25	3.267	0.748	3	3	3	0
0.30	3.319	0.819	3	3	3	0
0.35	3.376	0.892	3	3	3	0
0.40	3.437	0.971	3	3	3	0
0.45	3.501	1.042	3	3	3	0
0.50	3.567	1.111	3	3	3	0
0.55	3.613	1.159	3	3	5	2
0.60	3.681	1.229	3	3	5	2
0.65	3.744	1.290	3	3	5	2
0.70	3.816	1.354	3	3	5	2
0.75	3.890	1.421	3	3	5	2
0.80	3.960	1.483	3	3	5	2
0.85	4.026	1.535	3	3	5	2
0.90	4.105	1.597	3	3	5	2
0.95	4.176	1.654	3	3	5	2

A variety of tree sizes is desired so that the M-TARGET method is able to begin its search across different depths without having to rely solely on GROW and PRUNE operations to guide it to deeper or shallower trees. Since there are several settings that give a good distribution of starting tree sizes, $(\alpha, \beta) = (0.4, 0)$ is selected for simplicity as a competitor to the chosen default settings of $(\alpha, \beta) = (0.5, 2)$ in BTR (Chipman et al. 2002).

Figure 4.2.2.1. Distribution of Randomly Generated Trees for $(\alpha, \beta) = (0.4, 0)$

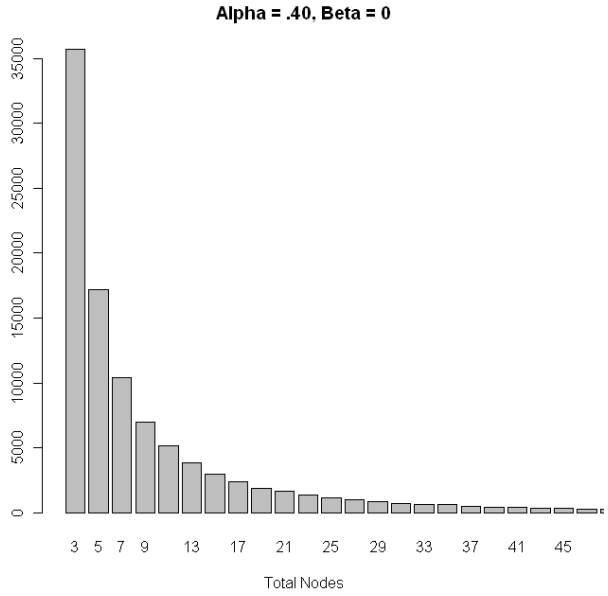
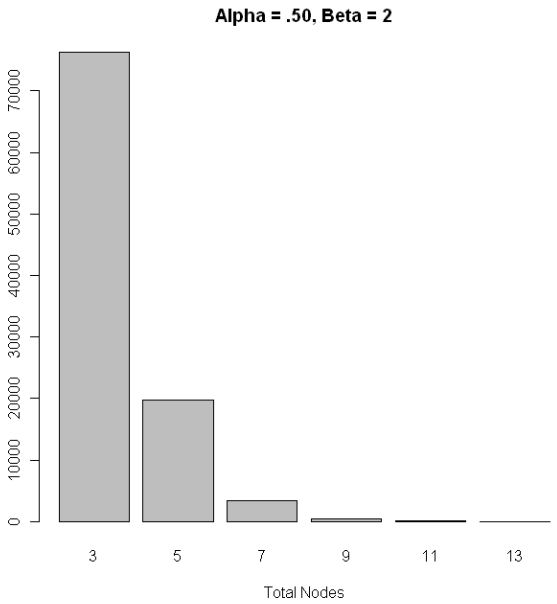


Figure 4.2.2.2. Distribution of Randomly Generated Trees for $(\alpha, \beta) = (0.5, 2)$



If p_{split} is constant $(\beta = 0)$, the expected size of the tree may also be calculated analytically from the extension of the Catalan number (Papagelis and Kalles 2001) where the number of possible binary trees is given by:

$$C(|T|) = \begin{cases} 0 & |T| = 0 \\ \frac{1}{|T|}\binom{2|T|-2}{|T|-1} & |T| > 0 \end{cases}, \tag{4.2.1}$$

where $C(|T|)$ is the number of topologically different trees having $|T|$ terminal nodes. The probability of growing a binary decision tree with $|T|$ terminal nodes is then:

$$P(|T|) = \begin{cases} 0 & |T| \le 1 \\ C(|T|) \times p_{split}{}^{|T|-2} \times (1 - p_{split})^{|T|} & |T| > 1, \beta = 0 \end{cases}, \tag{4.2.2}$$

where it is assumed that there is at least one split present in the decision tree and that $\beta = 0$, making p_{split} constant and not varying with depth. The expected number of terminal nodes given a constant value of p_{split} is calculated to be:

$$E(|T|\,|\,p_{split}) = \sum_{t=1}^{\infty} (P(t) \times t). \tag{4.2.3}$$

4.2.3 Results and Analysis of Chosen M-TARGET Settings

A repeated measures ANOVA is used to compare the four combinations of settings on each of the five data sets. Each of the ten folds per data set is maintained throughout every run. The different combinations of M-TARGET settings are considered to be the repeated measures applied to the same ten folds. See Tables 4.2.3.1 – 4.2.3.5 for the results. The test set root mean squared error (*RMSE*) is significantly lower for the 5 runs of 2,000 generations than the 2 runs of 5,000 generations in the simple linear regression case for the Abalone and Plasma data sets,

F(1, 9) = 8.550, p = 0.017 and F(1, 9) = 6.351, p = 0.033, respectively. No other significant differences were found among the settings for the number of runs and generations, and settings for p_{split}.

The training set *BIC* is significantly lower for the 5 runs of 2,000 generations than the 2 runs of 5,000 generations in the simple linear regression case for the Abalone and Concrete data sets, F(1, 9) = 50.502, p = .000 and F(1, 9) = 9.804, p = 0.012, respectively. For the multiple regression case on the Boston Housing data, the training *BIC* values for the 2 runs of 5,000 generations were significantly lower than the 5 runs of 2,000 generations, F(1, 9) = 8.785, p = 0.016.

M-TARGET found identical trees for the four combinations of settings in the multiple regression case for the Abalone and Plasma Beta data sets. This would suggest that the run lengths in the multiple regression context are sufficiently long enough for M-TARGET to reach a consistently stable solution and that the choice of initial settings do not prevent the method from searching the same tree space and finding the same model. Additionally, M-TARGET found that the best tree model for the Plasma Beta data set is the root node. Partitioning the data further did not significantly improve training set accuracy.

M-TARGET performed better overall using the 5 runs of 2,000 generations than the 2 runs of 5,000 generations. The wider distribution of initial randomly generated trees for $(\alpha, \beta) = (0.4, 0)$ makes it more desirable than $(\alpha, \beta) = (0.5, 2)$ as M-TARGET does not need to rely solely on GROW operations to construct larger trees. For further use and comparisons among different methods, the 5 runs of 2,000 generations and $(\alpha, \beta) = (0.4, 0)$ are used as default settings.

Table 4.2.3.1. Mean M-TARGET Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Boston Housing Data (with standard deviations in parentheses)

	Simple Linear Regression		Multiple Linear Regression	
M-TARGET Settings: (α, β), Generations, Runs	*RMSE*	Terminal Nodes	*RMSE*	Terminal Nodes
(.4, 0), 2000, 5	0.179 (0.033)	13.100 (1.101)	0.163 (0.036)	3.100 (0.316)
(.5, 2), 2000, 5	0.189 (0.038)	12.300 (1.494)	0.186 (0.103)	3.100 (0.316)
(.4, 0), 5000, 2	0.191 (0.033)	14.600 (2.119)	0.188 (0.102)	3.200 (0.422)
(.5, 2), 5000, 2	0.191 (0.026)	14.000 (1.414)	0.186 (0.104)	3.000 (0.471)

N = 10 folds

Table 4.2.3.2. Mean M-TARGET Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Abalone Data (with standard deviations in parentheses)

	Simple Linear Regression		Multiple Linear Regression	
M-TARGET Settings: (α, β), Generations, Runs	*RMSE*	Terminal Nodes	*RMSE*	Terminal Nodes
(.4, 0), 2000, 5	2.266 (0.117)	16.100 (1.663)	2.152 (0.142)	2.900 (0.316)
(.5, 2), 2000, 5	2.249 (0.138)	16.000 (1.563)	2.152 (0.142)	2.900 (0.316)
(.4, 0), 5000, 2	2.304 (0.098)	19.200 (1.989)	2.152 (0.142)	2.900 (0.316)
(.5, 2), 5000, 2	2.278 (0.107)	18.300 (1.494)	2.152 (0.142)	2.900 (0.316)

N = 10 folds

Table 4.2.3.3. Mean M-TARGET Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Concrete Data (with standard deviations in parentheses)

	Simple Linear Regression		Multiple Linear Regression	
M-TARGET Settings: (α, β), Generations, Runs	*RMSE*	Terminal Nodes	*RMSE*	Terminal Nodes
(.4, 0), 2000, 5	6.770 (0.412)	26.200 (1.989)	5.771 (0.895)	10.400 (1.174)
(.5, 2), 2000, 5	6.995 (0.753)	25.500 (2.369)	5.819 (1.024)	10.400 (1.174)
(.4, 0), 5000, 2	6.657 (0.494)	29.600 (3.062)	5.936 (1.347)	11.100 (2.079)
(.5, 2), 5000, 2	6.791 (0.636)	31.100 (3.446)	5.304 (0.879)	12.400 (1.776)

N = 10 folds

Table 4.2.3.4. Mean M-TARGET Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the PM10 Data (with standard deviations in parentheses)

	Simple Linear Regression		Multiple Linear Regression	
M-TARGET Settings: (α, β), Generations, Runs	*RMSE*	Terminal Nodes	*RMSE*	Terminal Nodes
(.4, 0), 2000, 5	0.779 (0.107)	9.100 (1.663)	0.777 (0.062)	2.900 (0.316)
(.5, 2), 2000, 5	0.804 (0.097)	9.000 (2.261)	0.783 (0.057)	2.900 (0.316)
(.4, 0), 5000, 2	0.813 (0.124)	9.300 (2.163)	0.785 (0.054)	2.800 (0.422)
(.5, 2), 5000, 2	0.802 (0.115)	9.200 (0.789)	0.781 (0.049)	2.900 (0.316)

N = 10 folds

Table 4.2.3.5. Mean M-TARGET Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Plasma Beta Data (with standard deviations in parentheses)

	Simple Linear Regression		Multiple Linear Regression	
M-TARGET Settings: (α, β), Generations, Runs	*RMSE*	Terminal Nodes	*RMSE*	Terminal Nodes
(.4, 0), 2000, 5	213.148 (69.969)	8.100 (0.738)	166.416 (42.236)	1.000 (0.000)
(.5, 2), 2000, 5	200.020 (51.038)	8.200 (1.229)	166.416 (42.236)	1.000 (0.000)
(.4, 0), 5000, 2	221.954 (65.875)	8.400 (1.350)	166.416 (42.236)	1.000 (0.000)
(.5, 2), 5000, 2	229.276 (59.936)	8.300 (1.252)	166.416 (42.236)	1.000 (0.000)

N = 10 folds

4.3 Comparison Between M-TARGET and Treed Regression in the Simple Linear Regression Case

A paired samples T-test is used to compare the test *RMSE* results to M-TARGET and TR in the simple linear regression case. There is a significant difference in the mean PM10 test set *RMSE* between M-TARGET (M = 0.779, SD = 0.107) and TR (M = 0.827, SD = 0.118); t(9) = -2.947, p = 0.016, but no other significant differences were found in the other four data sets.

M-TARGET produced comparable test set results to TR using smaller trees. See tables 4.3.1 – 4.3.5 for the results. If accuracy is comparable, smaller trees are more desirable than larger ones because they are less complex and easier to interpret. In four of the five data sets, M-TARGET produced smaller trees than TR on average and significantly smaller trees in two of them. In the Abalone data set, tree sizes from M-TARGET (M = 16.100, SD = 1.663) are significantly smaller than TR (M = 28.200, SD = 2.300); t(9) = -14.034, p = 0.000. In the

Concrete data set, tree sizes from M-TARGET (M = 26.200, SD = 1.989) are significantly smaller than TR (M = 34.200, SD = 2.658); t(9) = -6.197, p = 0.000. TR (M = 4.300, SD = 2.214) also produced significantly smaller trees than M-TARGET (M = 8.100, SD = 0.738) in the Plasma Beta data set; t(9) = 6.862, p = 0.000.

Table 4.3.1. Mean Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Boston Housing Data in the Simple Linear Regression Case (with standard deviations in parentheses)

Method	*RMSE*	Terminal Nodes
M-TARGET	0.179 (0.033)	13.100 (1.101)
Treed Regression	0.177 (0.030)	14.000 (2.494)
Difference (p-value)	0.002 (0.800)	-0.900 (0.287)

N = 10 folds

Table 4.3.2. Mean Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Abalone Data in the Simple Linear Regression Case (with standard deviations in parentheses)

Method	*RMSE*	Terminal Nodes
M-TARGET	2.266 (0.117)	16.100 (1.663)
Treed Regression	2.354 (0.192)	28.200 (2.300)
Difference (p-value)	-0.088 (0.069)	-12.100 (0.000)

N = 10 folds

Table 4.3.3. Mean Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Concrete Data in the Simple Linear Regression Case (with standard deviations in parentheses)

Method	*RMSE*	Terminal Nodes
M-TARGET	6.770 (0.412)	26.200 (1.989)
Treed Regression	6.547 (0.836)	34.200 (2.658)
Difference (p-value)	0.223 (0.370)	-8.000 (0.000)

N = 10 folds

Table 4.3.4. Mean Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the PM10 Data in the Simple Linear Regression Case (with standard deviations in parentheses)

Method	*RMSE*	Terminal Nodes
M-TARGET	0.779 (0.107)	9.100 (1.663)
Treed Regression	0.827 (0.118)	10.100 (3.348)
Difference (p-value)	-0.048 (0.016)	-1.000 (0.293)

N = 10 folds

Table 4.3.5. Mean Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Plasma Data in the Simple Linear Regression Case (with standard deviations in parentheses)

Method	*RMSE*	Terminal Nodes
M-TARGET	213.148 (63.969)	8.100 (0.738)
Treed Regression	198.280 (81.481)	4.300 (2.214)
Difference (p-value)	14.867 (0.632)	3.800 (0.000)

N = 10 folds

4.4 Comparisons Among M-TARGET, Treed Regression, and Bayesian Treed Regression in the Multiple Linear Regression Case

BTR is a close competitor to M-TARGET in that the tree space is searched stochastically, rather than greedily. A key difference between the two methods is that M-TARGET uses a group of trees in a "forest" to simultaneously search the tree space, while BTR searches using one tree at a time. Furthermore, M-TARGET utilizes genetic algorithms to guide the evolution process while BTR uses Monte Carlo Markov Chains in the form of the Metropolis-Hastings algorithm. The same default settings from Chipman et al. (2002) are used for the implementation of BTR: $(\alpha, \beta) = (0.5, 2)$ and 4,000 steps per run. Additionally, a burn-in time of 1,000 steps and 60 restarts per run are utilized to give BTR at least an equivalent number of searched trees so that a fair comparison to M-TARGET is made.

A repeated measures ANOVA is used to compare M-TARGET to TR and BTR in the multiple linear regression context. See tables 4.4.1 – 4.4.5 for the results along with the classical multiple linear regression. There is a significant difference in the Concrete test set *RMSE* among

the three methods, $F(2, 18) = 7.504$, $p = 0.004$. Post hoc Bonferroni tests show that the test set *RMSE* is significantly lower in M-TARGET ($M = 5.770$, $SD = 0.895$) than BTR ($M = 6.711$, $SD = 0.579$), $p = 0.005$. A significant difference is also found in the PM10 data set, $F(2, 18) = 13.026$, $p = 0.000$. Post hoc Bonferroni tests show that the test set *RMSE* is significantly lower in BTR ($M = 0.753$, $SD = 0.059$) than TR ($M = 0.799$, $SD = 0.045$), $p = 0.001$.

Additionally, M-TARGET and TR both produced smaller trees than BTR on the Abalone ($F(1.025, 9.222) = 76.850$, $p = 0.000$), PM10 ($F(1.258, 11.326) = 52.040$, $p = 0.000$), and Plasma ($F(2, 18) = 729.000$, $p = 0.000$) data sets. BTR produced smaller trees than the other two methods on the Concrete data set, $F(2, 18) = 16.308$, $p = 0.000$.

M-TARGET produced significantly better performing trees with respect to *RMSE* on one data set and smaller, more parsimonious trees than BTR in three of the remaining four data sets without sacrificing accuracy. BTR produced smaller trees than M-TARGET on the Concrete data set, though BTR had a significantly higher *RMSE* than the other two methods.

Table 4.4.1. Mean Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Boston Housing Data in the Multiple Linear Regression Case (with standard deviations in parentheses)

Method	*RMSE*	Terminal Nodes
Multiple Linear Regression	0.185 (0.035)	N/A
M-TARGET	0.163 (0.036)	3.100 (0.316)
Treed Regression	0.223 (0.117)	3.100 (0.568)
Bayesian Treed Regression	0.167 (0.037)	2.700 (0.949)
M-TARGET – TR Difference (p-value)	-0.060 (0.270)	0.000 (1.000)
M-TARGET – BTR Difference (p-value)	-0.004 (1.000)	0.400 (0.808)
TR – BTR Difference (p-value)	0.056 (0.372)	0.400 (0.808)

N = 10 folds

Table 4.4.2. Mean Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Abalone Data in the Multiple Linear Regression Case (with standard deviations in parentheses)

Method	*RMSE*	Terminal Nodes
Multiple Linear Regression	2.216 (0.124)	N/A
M-TARGET	2.152 (0.142)	2.900 (0.316)
Treed Regression	2.142 (0.119)	3.000 (0.000)
Bayesian Treed Regression	2.175 (0.099)	11.000 (2.906)
M-TARGET – TR Difference (p-value)	0.009 (0.996)	-0.100 (1.000)
M-TARGET – BTR Difference (p-value)	-0.024 (1.000)	-8.100 (0.000)
TR – BTR Difference (p-value)	-0.033 (0.667)	-8.000 (0.000)

N = 10 folds

Table 4.4.3. Mean Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Concrete Data in the Multiple Linear Regression Case (with standard deviations in parentheses)

Method	*RMSE*	Terminal Nodes
Multiple Linear Regression	10.467 (0.662)	N/A
M-TARGET	5.770 (0.895)	10.400 (1.174)
Treed Regression	5.980 (0.734)	9.300 (2.214)
Bayesian Treed Regression	6.711 (0.579)	6.600 (0.699)
M-TARGET – TR Difference (p-value)	-0.209 (1.000)	1.100 (0.600)
M-TARGET – BTR Difference (p-value)	-0.940 (0.005)	3.800 (0.000)
TR – BTR Difference (p-value)	-0.731 (0.053)	2.700 (0.019)

N = 10 folds

Table 4.4.4. Mean Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the PM10 Data in the Multiple Linear Regression Case (with standard deviations in parentheses)

Method	*RMSE*	Terminal Nodes
Multiple Linear Regression	0.817 (0.064)	N/A
M-TARGET	0.777 (0.062)	2.900 (0.316)
Treed Regression	0.799 (0.045)	2.500 (0.527)
Bayesian Treed Regression	0.753 (0.059)	6.100 (1.370)
M-TARGET – TR Difference (p-value)	-0.023 (0.101)	0.400 (0.110)
M-TARGET – BTR Difference (p-value)	0.024 (0.121)	-3.200 (0.000)
TR – BTR Difference (p-value)	0.046 (0.001)	-3.600 (0.000)

N = 10 folds

Table 4.4.5. Mean Test *RMSE* and Counts of Terminal Nodes for the 10-fold Cross Validations on the Plasma Data in the Multiple Linear Regression Case (with standard deviations in parentheses)

Method	*RMSE*	Terminal Nodes
Multiple Linear Regression	166.416 (42.236)	N/A
M-TARGET	166.416 (42.236)	1.000 (0.000)
Treed Regression	166.416 (42.236)	1.000 (0.000)
Bayesian Treed Regression	163.632 (46.287)	5.500 (0.527)
M-TARGET – TR Difference (p-value)	0.000 (N/A)	0.000 (N/A)
M-TARGET – BTR Difference (p-value)	2.784 (1.000)	-4.500 (0.000)
TR – BTR Difference (p-value)	2.784 (1.000)	-4.500 (0.000)

N = 10 folds

CHAPTER 5

DISCUSSION AND CONCLUSIONS

5.1 Summary

Modern data mining techniques allow data to be modeled in a meaningful way so that knowledge can be extracted and applied to solve real world problems. Decision trees are a data mining tool that creates flowchart-like visual models to easily communicate patterns and information from the data. Relatively heterogeneous data sets are recursively partitioned into smaller, more homogeneous sets where generalizations can be made to understand the interactions between predictors and how they relate to the target variable. The rules for partitioning are easily seen and interpreted in the tree structure, which is a major strength of decision tree modeling over other methods.

Improvements to the original CART decision tree modeling algorithm include the addition of leaf node models in Treed Regression to improve predictive accuracy and the incorporation of non-greedy tree induction methods in TARGET and Bayesian Treed to shift the focus from sequentially adding locally optimal splits to maximizing global optimality instead. These alternative methods take into the account the possibility that the next best split in the data may not be locally optimal but can enable an overall better split partition to be created deeper in the tree structure. TARGET uses genetic algorithms to allow the tree space to be searched more

efficiently by using a "forest" of randomly modified trees while Bayesian Treed incorporates the Metropolis-Hastings algorithm to make modifications to a single tree at a time.

M-TARGET is a stochastically-based tree modeling algorithm that corrects problems in greedily based methods and incorporates models in the leaf nodes to improve predictive accuracy. M-TARGET is able to produce smaller decision trees without compromising accuracy because the complexity of the model is divided between the tree structure and the leaf node model. The genetic algorithms add great value to tree modeling because they allow the tree space to be searched more effectively as information can be shared amongst trees in the forest. Additionally, the genetic algorithms provide the flexibility to modify previously determined splits, putting the focus on global optimality rather than following a sequence of locally optimal splits as traditional greedily-based tree algorithms do. This allows for alternative fitness measures to be used, such as *BIC* or the C-statistic, the area under the Receiver Operating Characteristic (*ROC*) curve, in the case of a categorical target variable. The randomly generated splits also remove the bias in variable selection.

M-TARGET uses a "forest" of randomly generated trees to search the tree space. Each tree begins with a root node, where the probability of splitting is given by p_{split}. If split, a randomly generated split rule partitions the data and the process is repeated on the newly created child nodes until all are terminal or another stopping criterion is reached, such as maximum depth or maximum tree size. The data are partitioned according to the split rules and a regression model is built in each terminal node using its observations. The forest of trees is evolved across generations by applying genetic operations that create new trees from the current forest. Information is shared between trees via genetic algorithms to provide better performing offspring. This evolutionary process continues until the predetermined number of generations

has been reached, where the best performing tree in the final generation is selected as the champion.

Results in this dissertation show that M-TARGET is competitive with other methods, but is doing so with significantly smaller trees. Parsimonious trees are favorable to larger, more complex ones because they are easier to interpret and the focus is placed on the more important variables. Some of the results were not significantly different, but M-TARGET outperformed Treed Regression and Bayesian Treed Regression by either producing more accurate or smaller trees. The flexibility of the M-TARGET algorithm allows it to find the competitors' trees and search a broader area of the tree space for even better tree models.

More detailed and further comparisons to the other methods would be possible with additional computing power or a faster operating environment. The limitations of both the speed and memory of R presented challenges in making runs.

5.2 Limitations of the Study

5.2.1 Computation Time

The computation time required for the runs is a limiting factor in the R environment. A total of 10,000 generations took a minimum of 12 to 24 hours to complete, depending on the data set and computer. There did not appear to be any "memory effects"; it appeared that the run time is invariant to how it is divided, whether 5 runs of 2,000 generations or a single run of 10,000 generations. Larger data sets with more observations and variables were attempted, but runs were moving as slow as a rate of 26 generations per hour.

The extremely long run times made it difficult to make comparisons on additional or larger data sets. Each data set required four different combinations of settings, each consisting of

a combined 10,000 generations, on ten different folds for a total of 400,000 generations from M-TARGET alone. BTR and TR add further run time as well.

The BTR environment in R is limiting as well, as it did not have the same capacity to handle large data sets as M-TARGET did. M-TARGET would handle large data sets slowly while BTR developed run time errors and program crashes. On smaller data sets, due to the C++ extensions, BTR runs were very fast and not prohibitively long unlike M-TARGET.

5.2.2 *BIC* Penalty

The Bayesian Information Criterion (*BIC*) is a fitness measure that features a penalty for tree complexity. It is a function of the *SSE* and the number of parameters used in the model. The M-TARGET algorithm allows for any fitness measure to be used, but the *BIC* is selected because it favors smaller, more parsimonious models than the comparable *AIC*, and Cavanaugh and Neath (1999) suggested that the *BIC* asymptotically identifies the true model with probability one, provided that the family of candidate models under consideration includes the true model generating the data.

Further research is needed to identify the correct *BIC* penalty for M-TARGET. In several cases, M-TARGET outperformed TR in training, but had a higher test set *RMSE.* Further adjustment may be necessary to further improve the performance of M-TARGET.

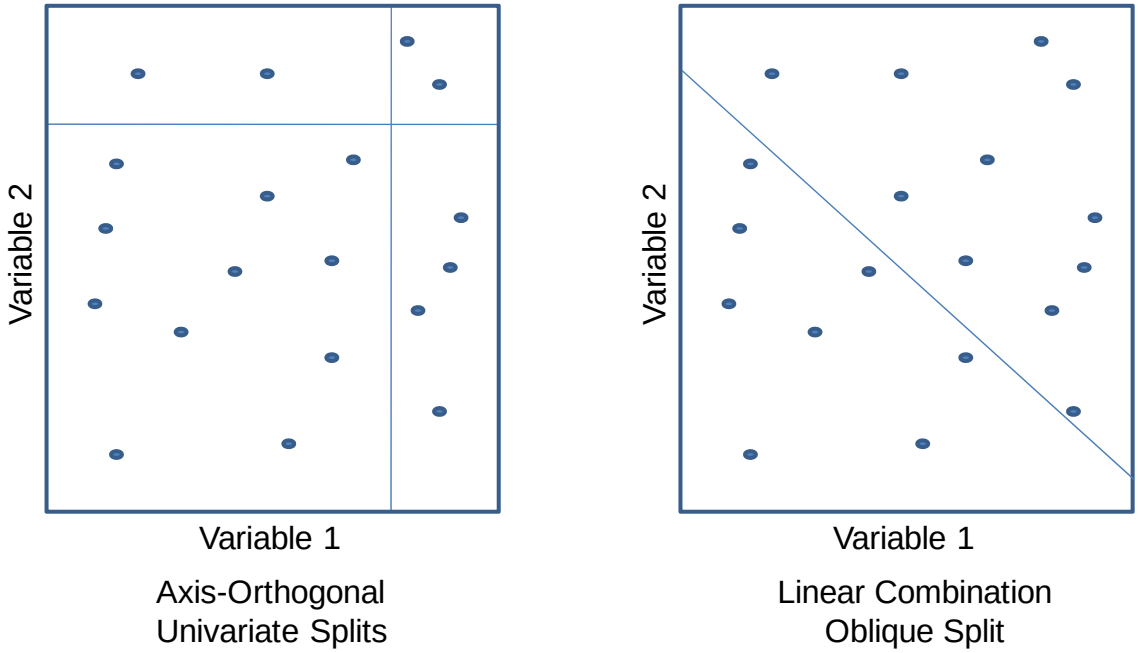
5.3 Future Research

M-TARGET is a strong algorithm, but it depends heavily on having computational resources to support it. M-TARGET could be recoded into C++ or Java in order to benefit from

a faster operating environment. Longer runs and additional restarts would be more feasible to execute if the computation time is greatly reduced.

M-TARGET may be extended to incorporate oblique splits rather than univariate axis-orthogonal splits. Oblique splits consist of linear combinations of predictor variables in order to make decision node splits more flexible and allow for finer interactions between variables to be represented in the splitting process. The addition of oblique splits into the tree structure should compress the extra information into each node. However, it will come at a cost in reduced interpretability.

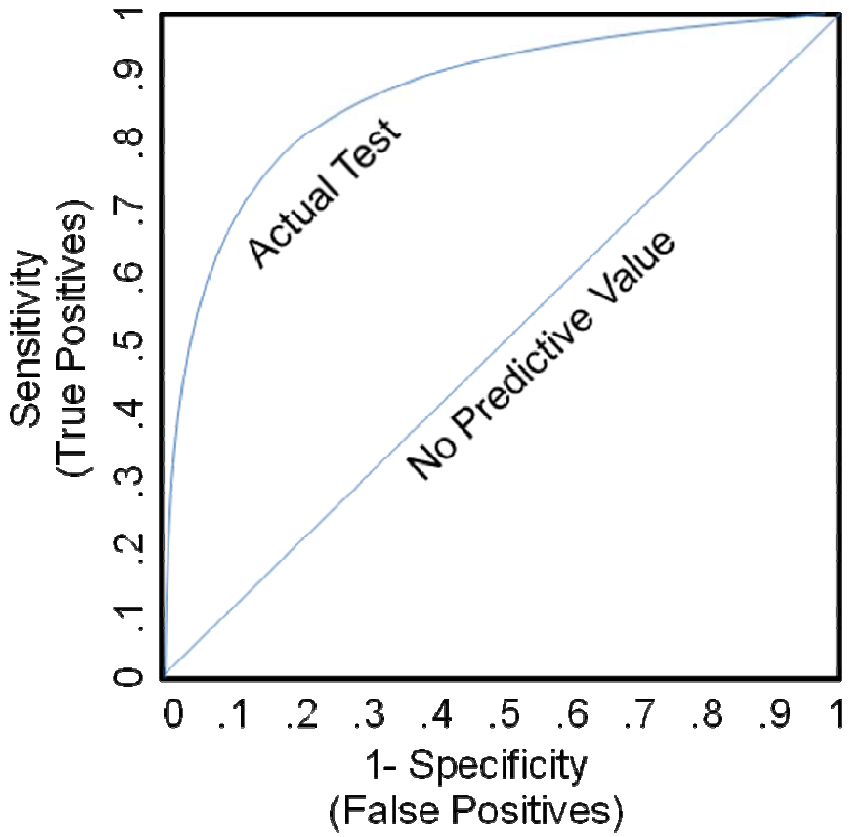
Figure 5.3.1. Oblique Split Example



M-TARGET can be extended to classification problems by using logistic regression models instead of linear regression models in the terminal nodes. Additionally, other classification-specific fitness measures, such as the C-statistic from the *ROC* curve may be

implemented to maximize application specific objectives in the case of binary outcomes. The *ROC* curve is a graph of the true positive versus false positive classifications for different cut-off points in determining whether to classify an observation as a "success" or "failure". The C-statistic is the area under the curve, which is sought to be maximized. Figure 5.3.1 gives an example diagram of an *ROC* curve. M-TARGET would be well equipped to handle these alternative fitness measures because they are created and utilized as the immediate objective function in the tree selection process, rather than serving as a diagnostic and reporting tool after the models have already been built and selected.

Figure 5.3.2. Example *ROC* Curve



REFERENCES

Akaike, H. (1974), "A New Look at the Statistical Model Identification," *IEEE Transactions on Automatic Control,* 19(6), 716-723.

Aldrin, M. (2004), "PM10.dat," *StatLib Datasets Archive*, Data file, retrieved from http://lib.stat.cmu.edu/datasets/PM10.dat

Alexander, W., and Grimshaw, S. (1996), "Treed Regression," *Journal of Computational and Graphical Statistics*, 5(2), 156-175.

Bala, J., Huang, J., Vafaie, H., Dejong, K., and Wechsler, H. (1995), "Hybrid Learning Using Genetic Algorithms and Decision Trees for Pattern Classification," in *14th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 719-724.

Bennett, K., and Mangasarian, O. (1992), "Robust Linear Programming Discrimination of Two Linearly Inseparable Sets," *Optimization Methods and Software*, 1, 23-34.

Breiman, L. (1996), "Heuristics of Instability and Stabilization in Model Selection," *The Annals of Statistics,* 24(6), 2350-2383.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984), *Classification and Regression Trees,* Boca Raton: Chapman & Hall/CRC Press.

Brodley, C., and Utgoff, P. (1995), "Multivariate Decision Trees," *Machine Learning*, 19, 45-77.

Buntine, W. (1992), "Learning Classification Trees," *Statistics and Computing,* 2, 63-73.

Cantu-Paz, E., and Kamath, C. (2003), "Inducing Oblique Decision Trees with Evolutionary Algorithms," *IEEE Transactions on Evolutionary Computation* 7(1), 54-68.

Cavanaugh, J. and Neath, A. (1999), "Generalizing the Derivation of the Schwarz Information Criterion," *Communications in Statistics – Theory and Methods,* 28(1), 49-66.

Chai, B., Huang, T., Zhuang, X., Yao, Y., and Sklansky, J. (1996), "Piecewise Linear Classifiers Using Binary Tree Structure and Genetic Algorithm," *Patten Recognition*, 29(11), 1905-1917.

Chan, K., and Loh, W. (2004), "LOTUS: An Algorithm for building Accurate and Comprehensible Logistic Regression Trees," *Journal of Computational and Graphical Statistics,* 13, 826-852.

Chan, P. (1989), "Inductive Learning with BCT," in *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 104-108.

Chaudhuri, P., Huang, M., Loh, W., and Yao, R. (1994), "Piecewise-Polynomial Regression Trees," *Statistica Sinica*, 4, 143-167.

Chaudhuri, P., Lo, W., Loh, W., Yang, C. (1995), "Generalized Regression Trees," *Statistica Sinica*, 5, 641-666.

Cherkauer, K., and Shavlik, J. (1996), "Growing Simpler Decision Trees to Facilitate Knowledge Discovery," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining,* pp. 315-318.

Chipman, H. (2009), Personal Communication.

Chipman, H., George, E., and McCulloch, R. (1998), "Bayesian CART Model Search," *American Statistical Association,* 93(443), 935-948.

----- (2000), "Hierarchical Priors for Bayesian CART Shrinkage," *Statistics and Computing*, 10, 17-24.

----- (2001), "The Practical Implementation of Bayesian Model Selection," *Model Selection*, ed. P. Lahiri, Beachwood, OH: Institute of Mathematical Statistics, pp. 65-116.

----- (2002), "Bayesian Treed Models," *Machine Learning*, 48, 299-320.

----- (2003), "Bayesian Treed Generalized Linear Models," *Bayesian Statistics 7*, eds. J. M. Bernardo, M. J. Bayarri, J. O. Berger, A. P. Dawid, D. Heckerman, A. F. M. Smith, and M. West, Oxford University Press.

Dalgaard, P. (2004), *Introductory Statistics with R*, New York, NY: Springer Science+Business Media, Inc.

De Jong, K. (1980), "Adaptive System Design: A Genetic Approach," *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-10(9), 566-574.

Denison, D., Mallick, B., and Smith, A. (1998), "A Bayesian CART Algorithm," *Biometrika*, 85(2), pp. 363-377.

Dobra, A., and Gehrke, J. (2002), "SECRET: A Scalable Linear Regression Tree Algorithm," in *In Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* pp. 481-487.

Elomaa, T., and Malinen, T. (2003), "On Lookahead Heuristics in Decision Tree Learning," in *Foundations of Intelligent Systems* (Vol. 2871/2003), ed. Zhong, N. et al., Berlin: Springer-Verlag, pp. 445-453.

Fan, G., and Gray, J. (2005), "Regression Tree Analysis Using TARGET," *Journal of Computational and Graphical Statistics,* 14(1), 206-218.

Folino, G., Pizzuti, C., and Spezzano, G. (1999), "A Cellular Genetic Programming Approach to Classification," in *Proceedings of the Genetic and Evolutionary Computation Conference 1999: Volume 2*, eds. W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Honavar, M. Jakiela, and R. Smith, pp. 1015-1020.

Frank, E., Wang, Y., Inglis, S., Holmes, G., and Witten, I. (1998), "Using Model Trees for Classification," *Machine Learning*, 32, 63-76.

Friedman, J. (1991), "Multivariate Adaptive Regression Splines," *The Annals of Statistics,* 19(1), 1-67.

----- (2006), "Recent Advances in Predictive (Machine) Learning," *Journal of Classification,* 23(2), 175-197.

Friedman, J., Hastie, T., and Tibshirani, R. (2000), "Additive Logistic Regression: A Statistical View of Boosting," *The Annals of Statistics*, 38(2), 337-374.

Gagné, P. and Dayton, C. M. (2002), "Best Regression Model Using Information Criteria," *Journal of Modern Applied Statistical Methods*, 1(2), 479-488.

Gray, J., and Fan, G. (2008), "Classification Tree Analysis Using TARGET," *Computational Statistics and Data Analysis,* 52, 1362-1372.

Green, P.J. (1995), "Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination," *Biometrika*, 82, 711-732.

Gramacy, R. (2007), "tgp: An R Package for Bayesian Nonstationary, Semiparametric Nonlinear Regression and Design by Treed Gaussian Process Models," *Journal of Statistical Software*, 19(9), 1-46.

----- (2010), Personal Communication.

Gramacy, R., and Taddy, M. (2010), "Categorical Inputs, Sensitivity Analysis, Optimization and Importance Tempering with tgp Version 2, an R Package for Treed Gaussian Process Models," *Journal of Statistical Software,* 33(6), 1-48.

Hand, D., Mannila, H., and Smyth, P. (2001), *Principles of Data Mining,* Cambridge, MA: The MIT Press.

Harrison, D. and Rubinfeld, D. (1978), "Housing Data Set," *UCI Machine Learning Repository*, Data file, retrieved from http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data.

Hartmann, C., Varshney, P., Mehrotra, K., and Gerberich, C. (1982), "Application of Information Theory to the Construction of Efficient Decision Trees," *IEEE Transactions on Information Theory*, IT-28(4), 565-577.

Heath, D., Kasif, S., and Salzberg, S. (1993), "Induction of Oblique Decision Trees," in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1002-1007.

Hinton, G., and Nowlan, S. (1987), "How Learning Can Guide Evolution," *Complex Systems*, 1, 495-502.

Holland, J. (1975), *Adaptation in Natural and Artificial Systems*, Cambridge, MA: The MIT Press.

Hothorn, T., Hornik, K., and Zeileis, A. (2006), "Unbiased Recursive Partitioning: A Conditional Inference Framework," *Journal of Computational and Graphical Statistics*, 15(3), 651-674.

Karalič, A. (1992), "Linear Regression in Regression Tree Leaves," Technical Report, Jožef Stefan Institute.

Karalič, A., and Cestnik, B. (1991), "The Bayesian Approach to Tree-Structured Regression," in *Proceedings of ITI-91*, Cavtat, Croatia.

Kass, G. (1980), "An Exploratory Technique for Investigating Large Quantities of Categorical Data," *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, 29(2), 119-127.

Kim, H., and Loh, W. (2001), "Classification Trees with Unbiased Multiway Splits," *Journal of the American Statistical Association*, 96, 598-604.

----- (2003), "Classification Trees with Bivariate Linear Discriminant Node Models," *Journal of Computational and Graphical Statistics*, 12, 512-530.

Koehler, A., and Murphree, E. (1988), "A Comparison of the Akaike and Schwarz Criteria for Selecting Model Order," *Applied Statistics*, 37(2), 187-195.

Kretowski, M. (2004), "An Evolutionary Algorithm for Oblique Decision Tree Induction," in *Lecture Notes in Computer Science* 3070, pp. 432-437.

Landwehr, N., Hall, M., and Frank, E. (2005), "Logistic Model Trees," *Machine Learning*, 59, 161-205.

Loh, W. (2002), "Regression Trees With Unbiased Variable Selection and Interaction Detection," *Statistica Sinica*, 12, 361-386.

Loh, W., and Shih, Y. (1997), "Split Selection Methods for Classification Trees," *Statistica Sinica,* 7, 815-840.

Loh, W., and Vanichsetakul, N. (1988), "Tree-Structured Classification Via Generalized Discriminant Analysis," *Journal of the American Statistical Association*, 83(403), 715-725.

Lubinsky, D. (1994), "Tree Structured Interpretable Regression," *Learning from Data: AI and Statistics*, eds. D. Fisher and H. Lenz, 112, 387-398.

Malerba, D., Esposito, F., Ceci, M., and Appice, A. (2004), "Top Down Induction of Model Trees with Regression and Splitting Nodes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(5), 612-625.

Mitchell, M. (1996), *An Introduction to Genetic Algorithms*. Cambridge, MA: The MIT Press.

Murthy, S., Kasif, S., and Salzberg, S. (1994), "A System for Induction of Oblique Decision Trees," *Journal of Artificial Intelligence Research*, 2, 1-32.

Murthy, S., Kasif, S., Salzberg, S., and Beigel, R. (1999), "OC1: Randomized Induction of Oblique Decision Trees," in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 322-327.

Murthy, S., and Salzberg, S. (1995a), "Decision Tree Induction: How Effective is the Greedy Heuristic?," *Proceedings of the First International Conference on Knowledge Discovery and Data Mining,* ed. Fayyad, U., and Uthurusamy, R., pp. 222-227.

Murthy, S., and Salzberg, S. (1995b), "Lookahead and Pathology in Decision Tree Induction," in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence,* pp. 1025-1031.

Nash, W., Sellers, T., Talbot, S., Cawthorn, A., and Ford, W. (1994), "Abalone Data Set," *UCI Machine Learning Repository,* Data file, retrieved from http://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data.

Neter, J., Kutner, M., Nachtsheim, C., and Wasserman, W. (1996), *Applied Linear Regression Models,* Chicago, IL: Irwin.

Nierenberg, D.W., Stukel, T.A., Baron, J.A., Dain, B.J., and Greenberg, E.R. (1989), "Determinants of Plasma Levels of Beta-Carotene and Retinol," *StatLib Datasets Archive*, Data file, retrieved from http://lib.stat.cmu.edu/datasets/Plasma_Retinol.

Nijssen, S. (2008), "Bayes Optimal Classification for Decision Trees," in *Proceedings of the 25th International Conference of Machine Learning*, pp. 696-703.

Norton, S. (1989), "Generating Better Decision Trees," in *In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence,* pp. 800-815.

Oh, S., Kim, C., and Lee, J. (2001), "Balancing the Selection Pressures and Migration Schemes in Parallel Genetic Algorithms for Planning Multiple Paths," in *Proceedings of the 2001 IEEE International Conference on Robotics and Automation,* pp. 3314-3319.

Pagallo, G. (1989), "Learning DNF by Decision Trees," in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence,* Detroit, Michigan: Morgan Kaufmann, pp. 639-644.

Papagelis, A., and Kalles, D. (2000), "GATree: Genetically Evolved Decision Trees," in *Proceedings of the 13th International Conference on Tools with Artificial Intelligence,* pp. 203-206.

-----(2001), "Breeding Decision Trees Using Evolutionary Techniques," in *Proceedings of the Eighteenth International Conference on Machine Learning*, Williamstown, MA, pp. 393-400.

Quinlan, J. (1986a), "Induction of Decision Trees," *Machine Learning*, 1, 86-106.

-----(1986b), "Simplifying Decision Trees," in *International Journal of Man-Machine Studies*, 27, 221-234.

-----(1992), "Learning with Continuous Classes," in *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*, pp. 1025-1031.

-----(1993), "C4.5: Programs for Machine Learning," Morgan Kaufmann Publishers.

R Development Core Team (2010), *R: A Language and Environment for Statistical Computing,* Vienna, Austria.

Schwarz, G. (1978), "Estimating the Dimension of a Model," *The Annals of Statistics*, 6(2), 461-464.

Shali, A., Kangavari, M., and Bina, B. (2007), "Using Genetic Programming for the Induction of Oblique Decision Trees," in *Proccedings of the Sixth International Conference on Machine Learning,* pp. 38-43.

Torgo, L. (1997a), "Functional Models for Regression Tree Leaves," in *Proceedings of the Fourteenth International Machine Learning Conference*, pp. 385-393.

----- (1997b), "Kernel Regression Trees," in *European Conference on Machine Learning*, Poster Paper.

Ulrich, K. (1986), "Servo Data Set," *UCI Machine Learning Repository*, Data file, retrieved from http://archive.ics.uci.edu/ml/machine-learning-databases/servo/servo.data.

Utgoff, P., and Brodley, C. (1990), "An Incremental Method for Finding Multivariate Splits for Decision Trees," in *Proceedings for the Seventh International Conference of Machine Learning,* pp. 58-65.

Vens, C., and Blockeel, H. (2006), "A Simple Regression Based Heuristic for Learning Model Trees," *Intelligent Data Analysis*, 10, 215-236.

Wang, Y., and Witten, I. (1997), "Inducing Model Trees for Continuous Classes," *In Proceedings of the 9th European Conference on Machine Learning Poster Papers,* eds. M. van Someren and G. Widmer, 128-137.

Yeh, I. (2007), "Concrete Compressive Strength Data Set," *UCI Machine Learning Repository,* Data file, retrieved from http://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength.

Zeileis, A., Hothorn, T., and Hornik, K. (2008), "Model-Based Recursive Partitioning," *Journal of Computational and Graphical Statistics,* 17(2), 492-514.

Zellner, A. (1971), *An Introduction to Bayesian Inference in Econometrics*, New York: John Wiley & Sons, Inc..

Zhu, M., and Chipman, H. (2006), "Darwinian Evolution in Parallel Universes: A Parallel Genetic Algorithm for Variable Selection," *Technometrics*, 48(4), 491-502.

APPENDIX

This appendix gives the M-TARGET source code written in R for all the comparison runs. There are several functions that control different processes of M-TARGET.

7.1 M-TARGETLinear Code

This function is the main driver of the program. It is the highest level function which accepts the user's arguments.

```
MTARGETLinear <- function(X, Y, alpha, beta, MinObsPerNode, numTrees, numGenerations,
maxDepth, simpleOrMultiple, obliqueSplits, fitnessMeasure)
{
  #create a matrix to hold the best fitness values
  fitnessTable <- matrix(0, nrow = numTrees, ncol = numGenerations)

  #declare the vector for the unsortedFitnessValues
  unsortedFitnessValues <- NULL

  #declare the vector for the sortedFitnessValues
  sortedFitnessValues <- NULL

  #create the initial forest
  Forest <- generationOneLinear(X, Y, alpha, beta, MinObsPerNode, numTrees, maxDepth,
  simpleOrMultiple, obliqueSplits, fitnessMeasure)

  #record the best trees
  bestTreesPerTNSize <- list()
  nullTree <- list(node = list(), fitness = 0.0, root = 1)

  bestTreesPerTNSize[[1]] <- nullTree

  #create a matrix to hold the values for the counts
  countTreesPerTNSize <- matrix(0, nrow=50)

  #record the best trees per terminal node size:
  #loop over all the trees in the forest:
  for(i in 1:length(Forest))
  {
    #get the length of the ith tree
    treeSize <- length(Forest[[i]]$node)

    #calculate the number of Terminal Nodes
    terminalNodeSize <- (treeSize + 1)/2

    #add the count to the matrix.
```

```
  countTreesPerTNSize[terminalNodeSize,1] <- countTreesPerTNSize[terminalNodeSize, 1]
  + 1

  #is this the biggest tree encountered?
  #if it is:
  if(length(bestTreesPerTNSize) < terminalNodeSize)
  {
    #create up to the point needed:
    for(k in (length(bestTreesPerTNSize) + 1):terminalNodeSize)
    {
      bestTreesPerTNSize[[k]] <- nullTree
    }

    #write in the tree:
    bestTreesPerTNSize[[terminalNodeSize]] <- Forest[[i]]
  }

  #if it isn't:
  else
  {
    #is there a tree there already?
    if(!is.null(bestTreesPerTNSize[[terminalNodeSize]]$fitness))
    {
      #does this tree beat the current best tree for this size?
      if(Forest[[i]]$fitness > bestTreesPerTNSize[[terminalNodeSize]]$fitness)
      {
        #need to replace the former best tree
        bestTreesPerTNSize[[terminalNodeSize]] <- Forest[[i]]
      }
    }

    #if not:
    else
    {
      #write in the tree:
      bestTreesPerTNSize[[terminalNodeSize]] <- Forest[[i]]
    }
  }
}


#get a list of the fitness values from the previous generation:
for(i in 1:length(Forest))
{
  unsortedFitnessValues[i] <- Forest[[i]]$fitness
}

#sort the fitness values
sortedFitnessValues <- rev(sort(unsortedFitnessValues))

for(i in 1:length(sortedFitnessValues))
{
  fitnessTable[i,1] <- sortedFitnessValues[i]
}

#subsequent generations:
for(j in 2:numGenerations)
{
  Forest <- forestNextGenLinear(Forest, X, Y, alpha, beta, MinObsPerNode, maxDepth,
  simpleOrMultiple, obliqueSplits, fitnessMeasure)

  #record the best trees per terminal node size:
  #loop over all the trees in the forest:
```

```
for(i in 1:length(Forest))
{
  #get the length of the ith tree
  treeSize <- length(Forest[[i]]$node)

  terminalNodeSize <- (treeSize + 1)/2

  #add the count to the matrix.
  countTreesPerTNSize[terminalNodeSize,1] <- countTreesPerTNSize[terminalNodeSize,
  1] + 1

  #is this the biggest tree encountered?
  #if it is:
  if(length(bestTreesPerTNSize) < terminalNodeSize)
  {
    #create up to the point needed:
    for(k in (length(bestTreesPerTNSize) + 1):terminalNodeSize)
    {
      bestTreesPerTNSize[[k]] <- nullTree
    }

    #write in the tree:
    bestTreesPerTNSize[[terminalNodeSize]] <- Forest[[i]]
  }

  #if it isn't:
  else
  {
    #is there a tree there already?
    if(!is.null(bestTreesPerTNSize[[terminalNodeSize]]$fitness))
    {
      #does this tree beat the current best tree for this size?
      if(Forest[[i]]$fitness > bestTreesPerTNSize[[terminalNodeSize]]$fitness)
      {
        #need to replace the former best tree
        bestTreesPerTNSize[[terminalNodeSize]] <- Forest[[i]]
      }
    }

    #if not:
    else
    {
      #write in the tree:
      bestTreesPerTNSize[[terminalNodeSize]] <- Forest[[i]]
    }
  }
}

#counter for the generation that just ran.
print(j)


#get a list of the fitness values from the previous generation:
for(i in 1:length(Forest))
{
  unsortedFitnessValues[i] <- Forest[[i]]$fitness
}

#sort the fitness values
sortedFitnessValues <- rev(sort(unsortedFitnessValues))

for(i in 1:length(sortedFitnessValues))
{
```

```
    fitnessTable[i,j] <- sortedFitnessValues[i]
   }

   gc()
  }

  #print(fitnessTable)

  bestTreesPerTNSize$countTreesPerTNSize <- countTreesPerTNSize

  #the return would be like a new forest
  output <- list(bestTreesPerTNSize, Forest)
  return(output)
}
```

7.2 generationOneLinear Code

This function controls the creation of the first generation of trees.

```
generationOneLinear <- function(X, Y, alpha, beta, MinObsPerNode, numTrees, maxDepth,
simpleOrMultiple, obliqueSplits, fitnessMeasure)
{
  #X is a list of the predictor variables
  #Y is a list of the response variables.

  #initialize a list of trees to serve as placeholders.
  forest <- list()

  #G1 has 50 trees in it.
  for(i in 1:numTrees)
  {
   #generate the tree
   Tree1 <- randomTree(X, alpha, beta, maxDepth, obliqueSplits)

   #plinko the observations
   Tree2 <- plinko(X, Tree1, MinObsPerNode, obliqueSplits)
   #make adjustments to trim the children
   Tree3 <- nodeChop(Tree2)
   #create a clean tree w/o the garbage nodes
   Tree4 <- treeCopy(Tree3)
   #apply the linear regression to the tree
   Tree5 <- linearRegression(X, Y, Tree4, simpleOrMultiple)
   #write the tree into the forest.
   forest[[i]] <- Tree5
  }
  return(forest)
}
```

7.3 forestNextGenLinear Code

This function controls the evolution process from generation to generation.

```
forestNextGenLinear <- function(Forest, X, Y, alpha, beta, MinObsPerNode, maxDepth,
simpleOrMultiple, obliqueSplits, fitnessMeasure)
{
  #clear out info things from previous generation that were created in the newForest
  process
```

```
for(i in 1:length(Forest))
{
  Forest[[i]]$sourceTree <- NULL
  Forest[[i]]$mutatedNode <- NULL
  Forest[[i]]$mutationType <- NULL
}

#the number of variables is the number of columns.
 numVars <- ncol(X)

#initialize a vector for the fitness of the previous generation's fitness values.
forest.fitness <- 0

#get a list of the fitness values from the previous generation:

#SSE:
if(fitnessMeasure == 0)
{
  for(i in 1:length(Forest))
  {
    forest.fitness[i] <- Forest[[i]]$fitness
  }
}

#BIC_1:
if(fitnessMeasure == 1)
{
  #get a list of the fitness values from the previous generation:
  for(i in 1:length(Forest))
  {
    forest.fitness[i] <- Forest[[i]]$BIC_1
  }
}

#BIC_2:
if(fitnessMeasure == 2)
{
  #get a list of the fitness values from the previous generation:
  for(i in 1:length(Forest))
  {
    forest.fitness[i] <- Forest[[i]]$BIC_2
  }
}

#AIC_1:
if(fitnessMeasure == 3)
{
  #get a list of the fitness values from the previous generation:
  for(i in 1:length(Forest))
  {
    forest.fitness[i] <- Forest[[i]]$AIC_1
  }
}

#AIC_2:
if(fitnessMeasure == 4)
{
  #get a list of the fitness values from the previous generation:
  for(i in 1:length(Forest))
  {
    forest.fitness[i] <- Forest[[i]]$AIC_2
  }
}
```

```
#initialize the next generation:
forestNextGeneration <- list()

#create a counter for the place in the next generation
forestNextGeneration.counter <- 1

#begin loop
#20 mutations, 5 clones
for(w in 1:20)
{
  chosen.genetic.operation <- sample(c(1:7), 1)

  ###############split set mutation (cutoff level only):

  if(chosen.genetic.operation == 1)
  {
    #draw one tree number to mutate
    temp.tree.number <- sample(length(Forest), 1, prob = rank(forest.fitness,
    ties.method = "average"))

    #draw the tree and make a temporary copy
    temp.tree <- Forest[[temp.tree.number]]

    number.of.nodes.in.temp <- length(temp.tree$node)

    #check to see if a singleton is drawn.
    if(number.of.nodes.in.temp < 2)
    {
      #keep track of the source tree
      temp.tree$sourceTree <- temp.tree.number

      #keep track of the mutation type
      temp.tree$mutationType <- "Ineligible Split Set Mutation"

      #copy the singleton into the next forest generation
      forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree

      #increment the counter for the position for the next tree to go into the forest
      forestNextGeneration.counter <- forestNextGeneration.counter + 1

    }

    #if it isn't a singleton drawn:
    else
    {
      #initialize a vector list of the non-terminal nodes
      non.terminal.nodes <- NULL
      non.terminal.nodes.num.obs <- NULL

      #create a list of nodes that are not terminal:
      #loop over all the nodes
      for(j in 1:number.of.nodes.in.temp)
      {
        #check to see if it is terminal:
        #new !is.null notation
        if(!is.null(temp.tree$node[[j]]$left))
        {
          #add to the list of non-terminal nodes.
          non.terminal.nodes <- c(non.terminal.nodes, j)
          non.terminal.nodes.num.obs <- c(non.terminal.nodes.num.obs,
          length(temp.tree$node[[j]]$obs))
        }
```

```
}

#pick a non-terminal node from the new tree to change the split rule
temp.node.number <- sample(non.terminal.nodes, 1, prob =
rank(non.terminal.nodes.num.obs, ties.method = "average"))

#write in the selected node to mutate into temp.node
temp.node <- temp.tree$node[[temp.node.number]]

#if it's a factor (categorical variable)
 if(temp.node$splitType)
{
  #random # of left/right method

  #draw the number of categories to put on the left
  #note that it's the number of levels - 1 because at least one has to go to the
  right.
  number.of.lefts <- sample(length(levels(X[[temp.node$splitVar]])) - 1, 1)

  #sample the number of lefts from the levels of the split variable
  left.child.levels <- sample(levels(X[[temp.node$splitVar]]),number.of.lefts)

  #write in the split set:
  sVal <- left.child.levels
}

#if not a factor or categorical variable
 if(temp.node$splitType == FALSE)
{
  #if no oblique splits
  if(obliqueSplits == 0 || length(temp.node$splitVar) == 1)
  {
    #draw one unique value from the continuous variable to be a split.
    #ok to draw any b/c the left path is <=
    sVal <- sample(unique(X[[temp.node$splitVar]]),1)
  }

  #if there is an oblique split:
  else
  {
    #draw one unique value from each of the continuous variables to be a split.
    #ok to draw any b/c the left path is <=
         sVal <- sample(unique(X[[temp.node$splitVar[1]]]),2)
    sVal <- c(sVal, sample(unique(X[[temp.node$splitVar[2]]]),2))
  }
 }

#write in the new variable value into the tree:
temp.tree$node[[temp.node.number]]$splitVal <- sVal

#note: at this point, the obs, obs counts, models, etc. are bad for temp.tree

#clear the things to be overwritten:
for(p in 1:length(temp.tree$node))
{
  temp.tree$node[[p]]$model <- NULL
  temp.tree$node[[p]]$SSE <- NULL
  temp.tree$node[[p]]$bestVar <- NULL
  temp.tree$node[[p]]$obs <- NULL
  temp.tree$node[[p]]$obsLength <- NULL
}

#need to re-run plinko on the tree
```

```
    temp.tree.plinkod <- plinko(X, temp.tree, MinObsPerNode, obliqueSplits)

    #need to re-run nodeChop on the tree
    temp.tree.chopped <- nodeChop(temp.tree.plinkod)

    #need to re-run treeCopy on the tree
    temp.tree.copied <- treeCopy(temp.tree.chopped)

    #need to re-run linearRegression on the tree
    temp.tree.reg <- linearRegression(X,Y, temp.tree.copied, simpleOrMultiple)

    #keep track of the source tree
    temp.tree.reg$sourceTree <- temp.tree.number

    #keep track of the original node changed
    temp.tree.reg$mutatedNode <- temp.node.number

    #keep track of the mutation type
    temp.tree.reg$mutationType <- "Split Set Mutation"

    #copy the tree into the next forest generation
    forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree.reg

    #increment the counter for the position for the next tree to go into the forest
    forestNextGeneration.counter <- forestNextGeneration.counter + 1
  }

#end of the split set mutation routine
}

#return(forestNextGeneration)

#for testing:
#treeTable(Forest[[temp.tree.number]])
#temp.node.number
#treeTable(temp.tree.reg)


################split rule mutation (variable and cutoff levels):

#add on top of what was previously done. Keep incrementing that nextgeneration
Counter

if(chosen.genetic.operation == 2)
{
  #draw one tree number to mutate
  temp.tree.number <- sample(length(Forest), 1, prob = rank(forest.fitness,
  ties.method = "average"))

  #draw the tree and make a temporary copy
  temp.tree <- Forest[[temp.tree.number]]

  number.of.nodes.in.temp <- length(temp.tree$node)

  #check to see if a singleton is drawn.
  if(number.of.nodes.in.temp < 2)
  {
    #keep track of the source tree
    temp.tree$sourceTree <- temp.tree.number

    #keep track of the mutation type
    temp.tree$mutationType <- "Ineligible Split Rule Mutation"
```

```
  #copy the singleton into the next forest generation
  forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree

  #increment the counter for the position for the next tree to go into the forest
  forestNextGeneration.counter <- forestNextGeneration.counter + 1
}

#if it isn't a singleton drawn:
else
{
  #initialize a vector list of the non-terminal nodes
  non.terminal.nodes <- NULL
  non.terminal.nodes.num.obs <- NULL

  #create a list of nodes that are not terminal:
  #loop over all the nodes
  for(j in 1:number.of.nodes.in.temp)
  {
    #check to see if it is terminal:
    if(!is.null(temp.tree$node[[j]]$left))
    {
      non.terminal.nodes <- c(non.terminal.nodes, j)
      non.terminal.nodes.num.obs <- c(non.terminal.nodes.num.obs,
      length(temp.tree$node[[j]]$obs))
    }
  }

  #pick a node from the new tree to change the split rule
  temp.node.number <- sample(non.terminal.nodes, 1, prob =
  rank(non.terminal.nodes.num.obs, ties.method = "average"))

  #I opted to drop the following because of potential reference issues.
  #write in the selected node to mutate into temp.node
  #temp.node <- temp.tree$node[[temp.node.number]]

  #prepare to draw another variable:
  #the number of variables is the number of columns.
  numVars <- ncol(X)

  #draw 1 variable index to split on from and write into svar

  #flag initialization for a categorical variable selected.
  variableTypes <- 0

  if(obliqueSplits == 0)
  {
    sVar <- sample(1:numVars,1)

    #check to see if the variable is a factor (True if it is)
    sType <- is.factor(X[[sVar]])
  }

  #if oblique splits are chosen:
  else
  {
    #draw two split variables
    sVar <- sample(1:numVars, 2)

    #check to see if they are both quantitative:

    if(is.factor(X[[sVar[1]]]))
    {
      variableTypes <- 1
```

```
  }

  if(is.factor(X[[sVar[2]]]))
  {
    variableTypes <- variableTypes + 1
  }

  #if one is qualitative:
  if(variableTypes >= 1)
  {
    #just pick one of the two variables and exit
    variableToPick <- sample(1:2,1)

    sVar <- sVar[variableToPick]

    #check to see if the variable is a factor (True if it is)
    sType <- is.factor(X[[sVar]])

  }

  #at this point, you may have a qual or quant variable.
  #variableTypes >= 1 mean that if it's a quant variable, then it's just a
  single variable split.

  if(variableTypes == 0)
  {
    #should return false:
         sType <- FALSE
  }
}

#write into the node
temp.tree$node[[temp.node.number]]$splitType <- sType

#write into the node
temp.tree$node[[temp.node.number]]$splitVar <- sVar


#if it's a factor or categorical variable
if(sType)
{
  #random # of left/right method

  #draw a number to put on the left
  #note that it's the number of levels - 1 because at least one has to go to the
  right.

  number.of.lefts <- sample(length(levels(X[[sVar]])) - 1, 1)

  #sample the number of lefts from the levels of the split variable
  left.child.levels <- sample(levels(X[[sVar]]),number.of.lefts)

  #write in the split set:
  sVal <- left.child.levels
 }

#if not a factor or categorical variable
else
{
  #if no oblique splits
  if(obliqueSplits == 0 || variableTypes != 0)
  {
    #draw one unique value from the continuous variable to be a split.
```

```
      #ok to draw any b/c the left path is <=
      sVal <- sample(unique(X[[sVar]]),1)
    }

    #if there is an oblique split:
    else
    {
      #draw one unique value from each of the continuous variables to be a split.
      #ok to draw any b/c the left path is <=
      temp1 <- sVar[1]
      temp2 <- sVar[2]
           sVal <- sample(unique(X[[temp1]]),2)
      sVal <- c(sVal, sample(unique(X[[temp2]]),2))
    }
  }


  #write in the new split value into the tree:
  temp.tree$node[[temp.node.number]]$splitVal <- sVal


  #note: at this point, the obs, obs counts, models, etc. are bad for temp.tree

  #clear the things to be overwritten:
  for(p in 1:length(temp.tree$node))
  {
    temp.tree$node[[p]]$model <- NULL
    temp.tree$node[[p]]$SSE <- NULL
    temp.tree$node[[p]]$bestVar <- NULL
    temp.tree$node[[p]]$obs <- NULL
    temp.tree$node[[p]]$obsLength <- NULL
  }


  #need to re-run plinko on the tree
  temp.tree.plinkod <- plinko(X, temp.tree, MinObsPerNode, obliqueSplits)

  #need to re-run nodeChop on the tree
  temp.tree.chopped <- nodeChop(temp.tree.plinkod)

  #need to re-run treeCopy on the tree
  temp.tree.copied <- treeCopy(temp.tree.chopped)

  #need to re-run linearRegression on the tree
  temp.tree.reg <- linearRegression(X,Y, temp.tree.copied, simpleOrMultiple)

  #keep track of the source tree
  temp.tree.reg$sourceTree <- temp.tree.number

  #keep track of the original node changed
  temp.tree.reg$mutatedNode <- temp.node.number

  #keep track of the mutation type
  temp.tree.reg$mutationType <- "Split Rule Mutation"

  #copy the tree into the next forest generation
  forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree.reg

  #increment the counter for the position for the next tree to go into the forest
  forestNextGeneration.counter <- forestNextGeneration.counter + 1
 }

#end of the split rule mutation routine
```

```
}


################node Swap mutation (trade nodes w/in same tree only):

if(chosen.genetic.operation == 3)
{
  #draw one tree number to mutate
  temp.tree.number <- sample(length(Forest), 1, prob = rank(forest.fitness,
  ties.method = "average"))

  #draw the tree and make a temporary copy
  temp.tree <- Forest[[temp.tree.number]]

  number.of.nodes.in.temp <- length(temp.tree$node)

  #initialize a vector list of the non-terminal nodes
  non.terminal.nodes <- NULL
  non.terminal.nodes.num.obs <- NULL

  #create a list of nodes that are not terminal:
  #loop over all the nodes
  for(j in 1:number.of.nodes.in.temp)
  {
    #check to see if it is terminal:
    #new !is.null notation
    if(!is.null(temp.tree$node[[j]]$left))
    {
      #add to the list of non-terminal nodes.
      non.terminal.nodes <- c(non.terminal.nodes, j)
      non.terminal.nodes.num.obs <- c(non.terminal.nodes.num.obs,
      length(temp.tree$node[[j]]$obs))
    }
  }

  #check to see if it has at least 2 splits in there somewhere:
  if(length(non.terminal.nodes) >= 2)
  {
    #pick 2 non-terminal nodes from the new tree to change the split rule
    #it's easier to just pull two at once into a vector than one at a time.
    temp.node.numbers <- sample(non.terminal.nodes, 2, prob =
    rank(non.terminal.nodes.num.obs, ties.method = "average"))

    #copy the selected node numbers for readability
    temp.node.one <- temp.node.numbers[1]
    temp.node.two <- temp.node.numbers[2]

    #copy the selected node split information for node one
    temp.node.one.splitType <- temp.tree$node[[temp.node.one]]$splitType
    temp.node.one.splitVar <- temp.tree$node[[temp.node.one]]$splitVar
    temp.node.one.splitVal <- temp.tree$node[[temp.node.one]]$splitVal

    #copy the selected node split information for node two
    temp.node.two.splitType <- temp.tree$node[[temp.node.two]]$splitType
    temp.node.two.splitVar <- temp.tree$node[[temp.node.two]]$splitVar
    temp.node.two.splitVal <- temp.tree$node[[temp.node.two]]$splitVal

    #trade node split information:
    temp.tree$node[[temp.node.one]]$splitType <- temp.node.two.splitType
    temp.tree$node[[temp.node.one]]$splitVar <- temp.node.two.splitVar
    temp.tree$node[[temp.node.one]]$splitVal <- temp.node.two.splitVal

    temp.tree$node[[temp.node.two]]$splitType <- temp.node.one.splitType
```

```
    temp.tree$node[[temp.node.two]]$splitVar <- temp.node.one.splitVar
    temp.tree$node[[temp.node.two]]$splitVal <- temp.node.one.splitVal


    #note: at this point, the obs, obs counts, models, etc. are bad for temp.tree

    #clear the things to be overwritten:
    for(p in 1:length(temp.tree$node))
    {
      temp.tree$node[[p]]$model <- NULL
      temp.tree$node[[p]]$SSE <- NULL
      temp.tree$node[[p]]$bestVar <- NULL
      temp.tree$node[[p]]$obs <- NULL
      temp.tree$node[[p]]$obsLength <- NULL
    }


    #need to re-run plinko on the tree
    temp.tree.plinkod <- plinko(X, temp.tree, MinObsPerNode, obliqueSplits)

    #need to re-run nodeChop on the tree
    temp.tree.chopped <- nodeChop(temp.tree.plinkod)

    #need to re-run treeCopy on the tree
    temp.tree.copied <- treeCopy(temp.tree.chopped)

    #need to re-run linearRegression on the tree
    temp.tree.reg <- linearRegression(X,Y, temp.tree.copied, simpleOrMultiple)

    #keep track of the source tree
    temp.tree.reg$sourceTree <- temp.tree.number

    #keep track of the original node changed
    temp.tree.reg$mutatedNode <- temp.node.numbers

    #keep track of the mutation type
    temp.tree.reg$mutationType <- "Node Swap"

    #copy the tree into the next forest generation
    forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree.reg
       }

  #if it does not have at least two splits, then it will just pass on the tree
  drawn, w/o mutation

  else
  {
    #keep track of the source tree
    temp.tree$sourceTree <- temp.tree.number

    #keep track of the mutation type
    temp.tree$mutationType <- "Ineligible Node Swap Mutation"

    #copy the tree into the next forest generation
    forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree
  }

  #increment the counter for the position for the next tree to go into the forest
  forestNextGeneration.counter <- forestNextGeneration.counter + 1

#end of the node swap mutation routine
}
```

```
############## transplant additions (new trees)

if(chosen.genetic.operation == 4)
{
  #create a new tree
  temp.tree <- randomTree(X, alpha, beta,  maxDepth, obliqueSplits)

  temp.tree.plinkod <- plinko(X, temp.tree, MinObsPerNode, obliqueSplits)

  temp.tree.chopped <- nodeChop(temp.tree.plinkod)

  temp.tree.copied <- treeCopy(temp.tree.chopped)

  temp.tree.reg <- linearRegression(X, Y, temp.tree.copied, simpleOrMultiple)

  #identify the new tree type
  temp.tree.reg$mutationType <- "Transplant"

  #copy the tree into the next forest generation
  forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree.reg

  #increment the counter for the position for the next tree to go into the forest
  forestNextGeneration.counter <- forestNextGeneration.counter + 1

#end of transplants
}

############## Grow (force a split on a tree)

if(chosen.genetic.operation == 5)
#or however many Grow operations there should be
{
  #copy over tree to grow

  #draw one tree number to mutate
  temp.tree.number <- sample(length(Forest), 1, prob = rank(forest.fitness,
  ties.method = "average"))

  #draw the tree and make a temporary copy
  temp.tree <- Forest[[temp.tree.number]]

  number.of.nodes.in.temp <- length(temp.tree$node)

  #initialize a vector list of the terminal nodes
  terminal.nodes <- NULL

  terminal.nodes.obs <- NULL

  #check to see if the maximum depth has been reached:

  #initialize Depth:
  depth <- 0

  #read all the depths
  for(i in 1:number.of.nodes.in.temp)
  {
    depth <- max(temp.tree$node[[i]]$depth, depth)
  }

  #if there are not too many nodes already:
  if(depth < maxDepth)
  {
```

```
#create a list of nodes that are terminal:
#loop over all the nodes
for(j in 1:number.of.nodes.in.temp)
{
  #check to see if it is terminal:
  #new !is.null notation
  if(is.null(temp.tree$node[[j]]$left))
  {
    #add to the list of terminal nodes.
    terminal.nodes <- c(terminal.nodes, j)

    #add to the list of terminal node obs the number of obs in the terminal node
    terminal.nodes.obs <- c(terminal.nodes.obs, length(temp.tree$node[[j]]$obs))
  }
}

#pick a node from the new tree to create the split rule
temp.node.number <- sample(terminal.nodes, 1, prob = rank(terminal.nodes.obs,
ties.method = "average"))

#prepare to draw another variable:
#the number of variables is the number of columns.
     numVars <- ncol(X)


#flag initialization for a categorical variable selected.
variableTypes <- 0

if(obliqueSplits == 0)
{
       sVar <- sample(1:numVars,1)

  #check to see if the variable is a factor (True if it is)
       sType <- is.factor(X[[sVar]])
}

#if oblique splits are chosen:
else
{
  #draw two split variables
  sVar <- sample(1:numVars, 2)

  #check to see if they are both quantitative:

  if(is.factor(X[[sVar[1]]]))
  {
    variableTypes <- 1
  }

  if(is.factor(X[[sVar[2]]]))
  {
    variableTypes <- variableTypes + 1
  }

  #if one is qualitative:
  if(variableTypes >= 1)
  {
    #just pick one of the two variables and exit
    variableToPick <- sample(1:2,1)

    sVar <- sVar[variableToPick]

    #check to see if the variable is a factor (True if it is)
```

```
    sType <- is.factor(X[[sVar]])

  }

  #at this point, you may have a qual or quant variable.
  #variableTypes >= 1 mean that if it's a quant variable, then it's just a
  single variable split.

  if(variableTypes == 0)
  {
    #should return false:
         sType <- FALSE
  }
}

#write into the node
temp.tree$node[[temp.node.number]]$splitType <- sType

#write into the node
temp.tree$node[[temp.node.number]]$splitVar <- sVar



#if it's a factor or categorical variable
       if(sType)
{
  #random # of left/right method

  #draw a number to put on the left
  #note that it's the number of levels - 1 because at least one has to go to the
  right.

  number.of.lefts <- sample(length(levels(X[[sVar]])) - 1, 1)

  #sample the number of lefts from the levels of the split variable
  left.child.levels <- sample(levels(X[[sVar]]),number.of.lefts)

  #write in the split set:
  sVal <- left.child.levels
}

#if not a factor or categorical variable
 else
{
  #if no oblique splits
  if(obliqueSplits == 0 || variableTypes != 0)
  {
    #draw one unique value from the continuous variable to be a split.
    #ok to draw any b/c the left path is <=
    sVal <- sample(unique(X[[sVar]]),1)
  }

  #if there is an oblique split:
  else
  {
    #draw one unique value from each of the continuous variables to be a split.
    #ok to draw any b/c the left path is <=
    temp1 <- sVar[1]
    temp2 <- sVar[2]
         sVal <- sample(unique(X[[temp1]]),2)
    sVal <- c(sVal, sample(unique(X[[temp2]]),2))
  }
 }
```

```
#write in the new split value into the tree:
temp.tree$node[[temp.node.number]]$splitVal <- sVal

#write in the child node information:
temp.tree$node[[temp.node.number]]$left <- number.of.nodes.in.temp + 1

temp.tree$node[[temp.node.number]]$right <- number.of.nodes.in.temp + 2

#create child nodes:

temp.tree$node[[number.of.nodes.in.temp + 1]] <- list(parent=NULL, left=NULL,
right=NULL, splitVar=NULL, splitType=NULL, splitVal=NULL, obs=NULL, obsLength =
NULL, model=NULL, depth=NULL)

temp.tree$node[[number.of.nodes.in.temp + 2]] <- list(parent=NULL, left=NULL,
right=NULL, splitVar=NULL, splitType=NULL, splitVal=NULL, obs=NULL, obsLength =
NULL, model=NULL, depth=NULL)

#write in parent information:
temp.tree$node[[number.of.nodes.in.temp + 1]]$parent <- temp.node.number

temp.tree$node[[number.of.nodes.in.temp + 2]]$parent <- temp.node.number

temp.tree$node[[number.of.nodes.in.temp + 1]]$depth <-
temp.tree$node[[temp.node.number]]$depth + 1

temp.tree$node[[number.of.nodes.in.temp + 2]]$depth <-
temp.tree$node[[temp.node.number]]$depth + 1

#note: at this point, the obs, obs counts, models, etc. are bad for temp.tree

#clear the things to be overwritten:
for(p in 1:length(temp.tree$node))
{
  temp.tree$node[[p]]$model <- NULL
  temp.tree$node[[p]]$SSE <- NULL
  temp.tree$node[[p]]$bestVar <- NULL
  temp.tree$node[[p]]$obs <- NULL
  temp.tree$node[[p]]$obsLength <- NULL
}


#need to re-run plinko on the tree
temp.tree.plinkod <- plinko(X, temp.tree, MinObsPerNode, obliqueSplits)

#need to re-run nodeChop on the tree
temp.tree.chopped <- nodeChop(temp.tree.plinkod)

#need to re-run treeCopy on the tree
temp.tree.copied <- treeCopy(temp.tree.chopped)

#need to re-run linearRegression on the tree
temp.tree.reg <- linearRegression(X,Y, temp.tree.copied, simpleOrMultiple)

#keep track of the source tree
temp.tree.reg$sourceTree <- temp.tree.number

#keep track of the original node changed
temp.tree.reg$mutatedNode <- temp.node.number

#keep track of the mutation type
temp.tree.reg$mutationType <- "Grow"
```

```
    #copy the tree into the next forest generation
    forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree.reg
  }

  #if the tree is too deep already, just pass it along:
  else
  {
    #keep track of the source tree
    temp.tree$sourceTree <- temp.tree.number

    #keep track of the mutation type
    temp.tree$mutationType <- "Ineligible Grow"

    forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree
  }

  #increment the counter for the position for the next tree to go into the forest
  forestNextGeneration.counter <- forestNextGeneration.counter + 1

#end of Grow
}


############## Prune (cut a split off a tree)

if(chosen.genetic.operation == 6)
#or however many Prune operations there should be
{
  #copy over tree to prune

  #draw one tree number to mutate
  temp.tree.number <- sample(length(Forest), 1, prob = rank(forest.fitness,
  ties.method = "average"))

  #draw the tree and make a temporary copy
  temp.tree <- Forest[[temp.tree.number]]

  number.of.nodes.in.temp <- length(temp.tree$node)

  if(number.of.nodes.in.temp < 4)
  {

    #keep track of the source tree
    temp.tree$sourceTree <- temp.tree.number

    #keep track of the mutation type
    temp.tree$mutationType <- "Ineligible Prune"

    #copy the singleton into the next forest generation
    forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree

  }

  else
  {
    #initialize a vector list of the terminal nodes
    terminal.nodes <- NULL

    #create a list of nodes that are terminal:
    #loop over all the nodes
    for(j in 1:number.of.nodes.in.temp)
    {
```

```
      #check to see if it is terminal:
      #new !is.null notation
      if(is.null(temp.tree$node[[j]]$left))
      {
        #add to the list of terminal nodes.
        terminal.nodes <- c(terminal.nodes, j)
      }
    }

    #pick a node (and child) from the new tree cut off
    temp.node.number <- sample(terminal.nodes, 1)

    #delete off the node and sibling:
    temp.tree$node[[temp.tree$node[[temp.node.number]]$parent]]$left <- NULL
    temp.tree$node[[temp.tree$node[[temp.node.number]]$parent]]$right <- NULL

    #need to re-run treeCopy on the tree
    temp.tree.copied <- treeCopy(temp.tree)

    #need to re-run linearRegression on the tree
    temp.tree.reg <- linearRegression(X,Y, temp.tree.copied, simpleOrMultiple)

    #keep track of the source tree
    temp.tree.reg$sourceTree <- temp.tree.number

    #keep track of the original node changed
    temp.tree.reg$mutatedNode <- temp.node.number

    #keep track of the mutation type
    temp.tree.reg$mutationType <- "Prune"

    #copy the tree into the next forest generation
    forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree.reg
  }

  #increment the counter for the position for the next tree to go into the forest
  forestNextGeneration.counter <- forestNextGeneration.counter + 1

#end of Prune
}


############## CrossOver (Trade nodes between two different trees)

if(chosen.genetic.operation == 7)
{
  #draw one tree number to mutate
  temp.tree.number.one <- sample(length(Forest), 1, prob = rank(forest.fitness,
  ties.method = "average"))

  #draw the tree and make a temporary copy
  temp.tree.one <- Forest[[temp.tree.number.one]]

  number.of.nodes.in.temp.one <- length(temp.tree.one$node)

  #draw second tree number to mutate
  temp.tree.number.two <- sample(length(Forest), 1, prob = rank(forest.fitness,
  ties.method = "average"))

  #draw the tree and make a temporary copy
  temp.tree.two <- Forest[[temp.tree.number.two]]

  number.of.nodes.in.temp.two <- length(temp.tree.two$node)
```

```
#just pass along:
if((number.of.nodes.in.temp.one < 4) || (number.of.nodes.in.temp.two < 4))
{
  #keep track of the source tree
  temp.tree.one$sourceTree <- temp.tree.number.one

  #keep track of the mutation type
  temp.tree.one$mutationType <- "Ineligible Crossover"

  #copy the singleton into the next forest generation
  forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree.one
}

else
{
  #initialize a vector list of the non-terminal nodes

  non.terminal.nodes.one <- NULL
  non.terminal.nodes.one.num.obs <- NULL

  #create a list of nodes that are not terminal:
  #loop over all the nodes
  for(j in 1:number.of.nodes.in.temp.one)
  {
    #check to see if it is terminal:
    #new !is.null notation
    if(!is.null(temp.tree.one$node[[j]]$left))
    {
      #add to the list of non-terminal nodes.
      non.terminal.nodes.one <- c(non.terminal.nodes.one, j)
      non.terminal.nodes.one.num.obs <- c(non.terminal.nodes.one.num.obs,
      length(temp.tree.one$node[[j]]$obs))
    }
  }

  #pick a non-terminal node from the new tree to change the split rule
  temp.node.number.one <- sample(non.terminal.nodes.one, 1, prob =
  rank(non.terminal.nodes.one.num.obs, ties.method = "average"))

  #copy the selected node split information for node one
  temp.node.one.splitType <- temp.tree.one$node[[temp.node.number.one]]$splitType
  temp.node.one.splitVar <- temp.tree.one$node[[temp.node.number.one]]$splitVar
  temp.node.one.splitVal <- temp.tree.one$node[[temp.node.number.one]]$splitVal

  non.terminal.nodes.two <- NULL
  non.terminal.nodes.two.num.obs <- NULL

  #create a list of nodes that are not terminal:
  #loop over all the nodes
  for(j in 1:number.of.nodes.in.temp.two)
  {
    #check to see if it is terminal:
    #new !is.null notation
    if(!is.null(temp.tree.two$node[[j]]$left))
    {
      #add to the list of non-terminal nodes.
      non.terminal.nodes.two <- c(non.terminal.nodes.two, j)
      non.terminal.nodes.two.num.obs <- c(non.terminal.nodes.two.num.obs,
      length(temp.tree.two$node[[j]]$obs))
    }
  }
```

```
#pick a non-terminal node from the new tree to change the split rule
temp.node.number.two <- sample(non.terminal.nodes.two, 1, prob =
rank(non.terminal.nodes.two.num.obs, ties.method = "average"))

#copy the selected node split information for node one
temp.node.two.splitType <- temp.tree.two$node[[temp.node.number.two]]$splitType
temp.node.two.splitVar <- temp.tree.two$node[[temp.node.number.two]]$splitVar
temp.node.two.splitVal <- temp.tree.two$node[[temp.node.number.two]]$splitVal


#trade node split information:
temp.tree.one$node[[temp.node.number.one]]$splitType <- temp.node.two.splitType
temp.tree.one$node[[temp.node.number.one]]$splitVar <- temp.node.two.splitVar
temp.tree.one$node[[temp.node.number.one]]$splitVal <- temp.node.two.splitVal

temp.tree.two$node[[temp.node.number.two]]$splitType <- temp.node.one.splitType
temp.tree.two$node[[temp.node.number.two]]$splitVar <- temp.node.one.splitVar
temp.tree.two$node[[temp.node.number.two]]$splitVal <- temp.node.one.splitVal

#note: at this point, the obs, obs counts, models, etc. are bad for temp.tree

#clear the things to be overwritten:
for(p in 1:length(temp.tree.one$node))
{
  temp.tree.one$node[[p]]$model <- NULL
  temp.tree.one$node[[p]]$SSE <- NULL
  temp.tree.one$node[[p]]$bestVar <- NULL
  temp.tree.one$node[[p]]$obs <- NULL
  temp.tree.one$node[[p]]$obsLength <- NULL
}

#clear the things to be overwritten:
for(p in 1:length(temp.tree.two$node))
{
  temp.tree.two$node[[p]]$model <- NULL
  temp.tree.two$node[[p]]$SSE <- NULL
  temp.tree.two$node[[p]]$bestVar <- NULL
  temp.tree.two$node[[p]]$obs <- NULL
  temp.tree.two$node[[p]]$obsLength <- NULL
}

#need to re-run plinko on the tree
temp.tree.one.plinkod <- plinko(X, temp.tree.one, MinObsPerNode, obliqueSplits)
temp.tree.two.plinkod <- plinko(X, temp.tree.two, MinObsPerNode, obliqueSplits)

#need to re-run nodeChop on the tree
temp.tree.one.chopped <- nodeChop(temp.tree.one.plinkod)
temp.tree.two.chopped <- nodeChop(temp.tree.two.plinkod)

#need to re-run treeCopy on the tree
temp.tree.one.copied <- treeCopy(temp.tree.one.chopped)
temp.tree.two.copied <- treeCopy(temp.tree.two.chopped)

#need to re-run linearRegression on the tree
temp.tree.one.reg <- linearRegression(X,Y, temp.tree.one.copied,
simpleOrMultiple)
temp.tree.two.reg <- linearRegression(X,Y, temp.tree.two.copied,
simpleOrMultiple)

#compare fitness values of the trees.

#if tree one beats tree two:
if(temp.tree.one.reg$TotalSSE <= temp.tree.two.reg$TotalSSE)
```

```
    {
      #keep track of the source tree
      temp.tree.one.reg$sourceTree <- temp.tree.number.one

      #keep track of the original node changed
      temp.tree.one.reg$mutatedNode <- temp.node.number.one

      #keep track of the mutation type
      temp.tree.one.reg$mutationType <- "Crossover"

      #copy the tree into the next forest generation
      forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree.one.reg
    }

    #if tree two beats tree one:
    else
    {
      #keep track of the source tree
      temp.tree.two.reg$sourceTree <- temp.tree.number.two

      #keep track of the original node changed
      temp.tree.two.reg$mutatedNode <- temp.node.number.two

      #keep track of the mutation type
      temp.tree.two.reg$mutationType <- "Crossover"

      #copy the tree into the next forest generation
      forestNextGeneration[[forestNextGeneration.counter]] <- temp.tree.two.reg

    }

   }

   forestNextGeneration.counter <- forestNextGeneration.counter + 1

 #end of crossover
}


 #end of loop for the 20 genetic operations
}

#end of all mutations

############## clones (best trees live on)

#have it fill in clones for the subtree swaps for now)

#forest.fitness holds the values of the fitness for the trees.
#make a copy of it
forest.fitness.copy <- forest.fitness

#sort the list in descending order
forest.fitness.copy.sorted <- rev(sort(forest.fitness.copy))

#create a vector to see what the best trees are
for(i in 1:5)
{
 #pulls the index of the tree in the forest which has the ith greatest fitness value
 tree.index.to.copy <- which(forest.fitness == forest.fitness.copy.sorted[i])[1]

 #copy the tree itself
 tree.to.copy <- Forest[[tree.index.to.copy]]
```

```
    #keep track of the mutation type
    tree.to.copy$mutationType <- "Clone"

    #copy over the best tree (remaining)
    forestNextGeneration[[forestNextGeneration.counter]] <- tree.to.copy

    #increment the counter for the position for the next tree to go into the forest
    forestNextGeneration.counter <- forestNextGeneration.counter + 1

  #end of clones
  }

  return(forestNextGeneration)


#end of newForest
}
```

7.4 randomTree Code

This function initializes each randomly generated tree.

```
randomTree <- function(X, alpha, beta, maxDepth, obliqueSplits)
{
  # X is a data frame containing all of the X variables
  # prob is the node splitting probability for growing a random tree

  #declare the tree to be a list consisting of nodes and fitness.
  #the node components are lists too.

  #fix the root node

  tree <- list(node=list(), fitness=0.0, root=1)

  #the number of variables is the number of columns.
  numVars <- ncol(X)

  #create a "placeholder" for future values.
  nullNode <- list(parent=NULL, left=NULL, right=NULL, splitVar=NULL, splitType=NULL,
  splitVal=NULL, obs=NULL, obsLength = NULL, model=NULL, depth=NULL)

  #start a counter for the index of the next available free node.
  #lastNode <- 0

  #counter for the node that is being examined.
        n <- 1

  # Always split the root node. Write in the placeholders
        tree$node[[n]] <- nullNode

  #counter for the last available node in the list.
        lastNode <- 1

  #write in the depth of the node
  tree$node[[n]]$depth <- 0

  #draw 1 variable index to split on from and write into svar
```

```
#flag initialization for a categorical variable selected.
variableTypes <- 0

if(obliqueSplits == 0)
{
  sVar <- sample(1:numVars,1)

  #check to see if the variable is a factor (True if it is)
  sType <- is.factor(X[[sVar]])
}

#if oblique splits are chosen:
else
{
  #draw two split variables
  sVar <- sample(1:numVars, 2)

  #check to see if they are both quantitative:

  if(is.factor(X[[sVar[1]]]))
  {
    variableTypes <- 1
  }

  if(is.factor(X[[sVar[2]]]))
  {
    variableTypes <- variableTypes + 1
  }

  #if one is qualitative:
  if(variableTypes >= 1)
  {
    #just pick one of the two variables and exit
    variableToPick <- sample(1:2,1)

    sVar <- sVar[variableToPick]

    #check to see if the variable is a factor (True if it is)
    sType <- is.factor(X[[sVar]])

  }

  #at this point, you may have a qual or quant variable.
  #variableTypes >= 1 mean that if it's a quant variable, then it's just a single
  variable split.

  if(variableTypes == 0)
  {
    #should return false:
    sType <- FALSE
  }
}

#write into the node
tree$node[[n]]$splitType <- sType

#write into the node
tree$node[[n]]$splitVar <- sVar


#make the parent of the root node = 0 (IDs it as a root also)
tree$node[[1]]$parent <- 0
```

```
#if it's a factor or categorical variable
      if(sType)
{
        #random # of left/right method

  #draw a number to put on the left
  #note that it's the number of levels - 1 because at least one has to go to the
  right.

  number.of.lefts <- sample(length(levels(X[[sVar]])) - 1, 1)

  #sample the number of lefts from the levels of the split variable
  left.child.levels <- sample(levels(X[[sVar]]),number.of.lefts)

  #write in the split set:
  sVal <- left.child.levels
}

#if not a factor or categorical variable
else
{
  #if no oblique splits
  if(obliqueSplits == 0 || variableTypes != 0)
  {
    #draw one unique value from the continuous variable to be a split.
    #ok to draw any b/c the left path is <=
    sVal <- sample(unique(X[[sVar]]),1)
  }

  #if there is an oblique split:
  else
  {
    #draw one unique value from each of the continuous variables to be a split.
    #ok to draw any b/c the left path is <=
    temp1 <- sVar[1]
    temp2 <- sVar[2]
         sVal <- sample(unique(X[[temp1]]),2)
    sVal <- c(sVal, sample(unique(X[[temp2]]),2))
  }
}

#write into the node
tree$node[[n]]$splitVal <- sVal

 #write in the next available index in for the left child
tree$node[[n]]$left <- lastNode + 1

#write in the placeholders for the left child of the root
tree$node[[lastNode+1]] <- nullNode

#write in the root node as the parent of the left child
 tree$node[[lastNode+1]]$parent <- n

#write in the depth of the left child
 tree$node[[lastNode+1]]$depth <- tree$node[[n]]$depth + 1

#write in the next available index in for the right child
 tree$node[[n]]$right <- lastNode + 2

#write in the placeholders for the right child of the root
 tree$node[[lastNode+2]] <- nullNode

#write in the root node as the parent of the right child
```

```
tree$node[[lastNode+2]]$parent <- n

#write in the depth for the right child
tree$node[[lastNode+2]]$depth <- tree$node[[n]]$depth + 1

#add child nodes to list of nodes queued to split
nodesToSplit <- c((lastNode+1):(lastNode+2))

#update lastNode number
lastNode <- lastNode + 2

#begin splitting the next node from the queue (left child goes first)
while(length(nodesToSplit) > 0)
{
  #select top node from list, write the index into n
  n <- nodesToSplit[1]

  # remove top node from list
  nodesToSplit <- nodesToSplit[-1]

  #check to see if it has reached max depth:
  if(tree$node[[n]]$depth < maxDepth)
  {
    # splitting this node
    #draw a random uniform number and check to see if it's less than the probability
    to split.
    if(runif(1) <= (alpha * ((1 + tree$node[[n]]$depth)^(-beta))))
    {
      #draw 1 variable index to split on from and write into svar

      #flag initialization for a categorical variable selected.
      variableTypes <- 0

      if(obliqueSplits == 0)
      {
        sVar <- sample(1:numVars,1)

        #check to see if the variable is a factor (True if it is)
        sType <- is.factor(X[[sVar]])
      }

      #if oblique splits are chosen:
      else
      {
        #draw two split variables
        sVar <- sample(1:numVars, 2)

        #check to see if they are both quantitative:

        if(is.factor(X[[sVar[1]]]))
        {
          variableTypes <- 1
        }

        if(is.factor(X[[sVar[2]]]))
        {
          variableTypes <- variableTypes + 1
        }

        #if one is qualitative:
        if(variableTypes >= 1)
        {
          #just pick one of the two variables and exit
```

```
    variableToPick <- sample(1:2,1)

    sVar <- sVar[variableToPick]

    #check to see if the variable is a factor (True if it is)
    sType <- is.factor(X[[sVar]])

  }

  #at this point, you may have a qual or quant variable.
  #variableTypes >= 1 mean that if it's a quant variable, then it's just a
  single variable split.

  if(variableTypes == 0)
  {
    #should return false:
         sType <- FALSE
  }
}

#write into the node
tree$node[[n]]$splitType <- sType

#write into the node
tree$node[[n]]$splitVar <- sVar

#if it's a factor or categorical variable
       if(sType)
{
       #random # of left/right method

  #draw a number to put on the left
  #note that it's the number of levels - 1 because at least one has to go to the
  right.

  number.of.lefts <- sample(length(levels(X[[sVar]])) - 1, 1)

  #sample the number of lefts from the levels of the split variable
  left.child.levels <- sample(levels(X[[sVar]]),number.of.lefts)

  #write in the split set:
  sVal <- left.child.levels
 }

#if not a factor or categorical variable
 else
 {
  #if no oblique splits
  if(obliqueSplits == 0 || variableTypes != 0)
  {
    #draw one unique value from the continuous variable to be a split.
    #ok to draw any b/c the left path is <=
    sVal <- sample(unique(X[[sVar]]),1)
  }

  #if there is an oblique split:
  else
  {
    #draw one unique value from each of the continuous variables to be a split.
    #ok to draw any b/c the left path is <=
    temp1 <- sVar[1]
    temp2 <- sVar[2]
    sVal <- sample(unique(X[[temp1]]),2)
```

```
          sVal <- c(sVal, sample(unique(X[[temp2]]),2))
        }
       }

       #write into the node
       tree$node[[n]]$splitVal <- sVal

       #write in the next available index in for the left child
       tree$node[[n]]$left <- lastNode + 1

       #write in the placeholders for the left child
       tree$node[[lastNode+1]] <- nullNode

       #write in the parent of the left child
       tree$node[[lastNode+1]]$parent <- n

       #write in the depth of the left child
       tree$node[[lastNode+1]]$depth <- tree$node[[n]]$depth + 1

       #write in the next available index in for the right child
       tree$node[[n]]$right <- lastNode + 2

       #write in the placeholders for the right child
       tree$node[[lastNode+2]] <- nullNode

       #write in the parent of the right child
       tree$node[[lastNode+2]]$parent <- n

       #write in the depth for the right child
       tree$node[[lastNode+2]]$depth <- tree$node[[n]]$depth + 1

       #add child nodes to list of nodes queued to split
       nodesToSplit <- c(nodesToSplit, (lastNode+1):(lastNode+2))

       #update lastNode number
       lastNode <- lastNode + 2
      }

    #end check to see if the depth is ok.
    }

  #end while
  }

  #output the tree
  return(tree)
}
```

7.5 plinko Code

This function recursively partitions the observations according to the split rules determined by the randomTree function.

```
plinko <- function(X, Tree, MinObsPerNode, obliqueSplits)
{
  # X is a data frame containing all of the X variables
  # Tree is the tree

  #write in the list of all the observations into the first node's observations
```

```
Tree$node[[1]]$obs <- c(1:nrow(X))

# add root node to list of nodes to process
nodesToProcess <- c(Tree$root)

#begin loop to sort
#while there are still nodes down the path that haven't been sorted yet
while(length(nodesToProcess) > 0)
{
  #write in the first node index to process into i.
  i <- nodesToProcess[1]

  #check to see if there are obs to divide up
  if(is.null(Tree$node[[i]]$obs) == FALSE)
  {
    #check to see if there are enough obs to divide up:
    if(length(Tree$node[[i]]$obs) >= 2 * MinObsPerNode)
    {
      # check to see if it's NOT a terminal node (has children)
      if(is.null(Tree$node[[i]]$left) == FALSE)
      {
        #add the child nodes to split (divide) later.
        nodesToProcess <- c(nodesToProcess, Tree$node[[i]]$left, Tree$node[[i]]$right)

        #remove the top node from the queue
        nodesToProcess <- nodesToProcess[-1]

        # if is a categorical variable and factor split
        if(Tree$node[[i]]$splitType == TRUE)
        {
          #write in the left child node index into j for ease of reading
          j <- Tree$node[[i]]$left

          #write in the right child node index into k for ease of reading
          k <- Tree$node[[i]]$right

          #write in the obs that are both in the current node and a subset of those
          that are in the split set

          obs.for.left <- intersect(Tree$node[[i]]$obs,
          which(is.element(X[,Tree$node[[i]]$splitVar], Tree$node[[i]]$splitVal)))

          #write in the obs that are both in the current node and NOT in the left
          Child

          obs.for.right <- setdiff(Tree$node[[i]]$obs, obs.for.left)

          #check to see if there are enough obs per side:
          #if there aren't for the left, move them to the right.

          if(is.null(obs.for.left) == FALSE)
          {
            if(length(obs.for.left) < MinObsPerNode)
            {
              obs.for.right <- c(obs.for.right, obs.for.left)
              #There will be wrong values in the node, but it will be taken care of by
              TreeCleanUp.

              #delete the obs on the left.
              obs.for.left <- NULL
            }
          }
```

```
  #if there aren't for the right, move them to the left.
  #if both were too small, the right should be big enough for now.

  if(is.null(obs.for.right) == FALSE)
  {
    if(length(obs.for.right) < MinObsPerNode)
    {
      obs.for.left <- c(obs.for.left, obs.for.right)
      #There will be wrong values in the node, but it will be taken care of by
      TreeCleanUp.

      #delete the obs on the right.
      obs.for.right <- NULL
    }
  }

  #write in the nodes:
  Tree$node[[j]]$obs <- obs.for.left
  Tree$node[[k]]$obs <- obs.for.right
}

# quant split
else
{
  #check to see if it's a single variable split:

  #if no oblique splits:
  if(length(Tree$node[[i]]$splitVal) == 1)
  {
    #write in the left child node index into j for ease of reading
    j <- Tree$node[[i]]$left

    #write in the right child node index into k for ease of reading
    k <- Tree$node[[i]]$right

    #write in the obs that are both in the current node and less than or equal
    to the split value

    obs.for.left <- intersect(Tree$node[[i]]$obs,
    which(X[,Tree$node[[i]]$splitVar] <= Tree$node[[i]]$splitVal))

    if(length(obs.for.left) == 0)
    {
      obs.for.left <- NULL  # is this how to handle this situation?
    }

    #write in the obs that are both in the current node and NOT in the left
    Child

    obs.for.right <- setdiff(Tree$node[[i]]$obs, obs.for.left)

    #check to see if there are enough obs per side:

    #if there aren't for the left, move them to the right.
    if(is.null(obs.for.left) == FALSE)
    {
      if(length(obs.for.left) < MinObsPerNode)
      {
        obs.for.right <- c(obs.for.right, obs.for.left)
        #There will be wrong values in the node, but it will be taken care of
        by TreeCleanUp.

        #delete the obs on the left.
```

```
      obs.for.left <- NULL
    }
  }

  #if there aren't for the right, move them to the left.
  #if both were too small, the right should be big enough for now.

  if(is.null(obs.for.right) == FALSE)
  {
    if(length(obs.for.right) < MinObsPerNode)
    {
      obs.for.left <- c(obs.for.left, obs.for.right)
      #There will be wrong values in the node, but it will be taken care of
      by TreeCleanUp.

      #delete the obs on the right.
      obs.for.right <- NULL
    }
  }

  #write in the nodes:
  Tree$node[[j]]$obs <- obs.for.left
  Tree$node[[k]]$obs <- obs.for.right
}

#if oblique split is found:
else
{

  #write in the left child node index into j for ease of reading
  j <- Tree$node[[i]]$left

  #write in the right child node index into k for ease of reading
  k <- Tree$node[[i]]$right


  #write in the obs that are both in the current node and less than or equal
  to the split value
  slope <- (Tree$node[[i]]$splitVal[2] –
  Tree$node[[i]]$splitVal[1])/(Tree$node[[i]]$splitVal[4] –
  Tree$node[[i]]$splitVal[3])

  RHS <- (slope * (X[,Tree$node[[i]]$splitVar[2]] -
  Tree$node[[i]]$splitVal[3])) + Tree$node[[i]]$splitVal[1]

  obs.for.left <- intersect(Tree$node[[i]]$obs,which(RHS <=
  X[,Tree$node[[i]]$splitVar[1]]) )

  if(length(obs.for.left) == 0)
  {
    obs.for.left <- NULL  # is this how to handle this situation?
  }

  #write in the obs that are both in the current node and NOT in the left
  Child

  obs.for.right <- setdiff(Tree$node[[i]]$obs, obs.for.left)

  #check to see if there are enough obs per side:

  #if there aren't for the left, move them to the right.
  if(is.null(obs.for.left) == FALSE)
  {
```

```
          if(length(obs.for.left) < MinObsPerNode)
          {
            obs.for.right <- c(obs.for.right, obs.for.left)
            #There will be wrong values in the node, but it will be taken care of
            by TreeCleanUp.

            #delete the obs on the left.
            obs.for.left <- NULL
          }
        }

        #if there aren't for the right, move them to the left.
        #if both were too small, the right should be big enough for now.

        if(is.null(obs.for.right) == FALSE)
        {
          if(length(obs.for.right) < MinObsPerNode)
          {
            obs.for.left <- c(obs.for.left, obs.for.right)
            #There will be wrong values in the node, but it will be taken care of
            by TreeCleanUp.

            #delete the obs on the right.
            obs.for.right <- NULL
          }
        }

        #write in the nodes:
        Tree$node[[j]]$obs <- obs.for.left
        Tree$node[[k]]$obs <- obs.for.right
      }
    }

    #I don't think this is necessary:
    #if nothing goes one way or the other, delete the obs list in the empty nodes
    if(length(Tree$node[[j]]$obs) == 0)
    {
      Tree$node[[j]]$obs <- NULL
    }

    if(length(Tree$node[[k]]$obs) == 0)
    {
      Tree$node[[k]]$obs <- NULL
    }
  }

  # if it is a terminal node
  else
  {
    #remove the top node from the queue
    nodesToProcess <- nodesToProcess[-1]
  }
 }

 #if there are not enough obs to divide up
 else
 {
   #remove the top node from the queue
   nodesToProcess <- nodesToProcess[-1]
 }
}

#if there are no obs to divide up
```

```
    else
    {
      #remove the top node from the queue
      nodesToProcess <- nodesToProcess[-1]
    }

  }

  #input the number of obs per node
  for(i in 1:length(Tree$node))
  {
    #check to make sure that it's not empty:
    if(is.null(Tree$node[[i]]$obs) == FALSE)
    {
      Tree$node[[i]]$obsLength <- length(Tree$node[[i]]$obs)
    }
  }

  return(Tree)

}
```

7.6 nodeChop Code

This function removes nodes that do not meet the minimum number of observations specified in the arguments for MTARGETLinear.

```
nodeChop <- function(tree)
{
  #loop over all the nodes
  for(i in 1:length(tree$node))
  {
    #check to see if the LC or RC is empty

    #get the LC and RC info, if it exists:
    if(is.null(tree$node[[i]]$left) == FALSE)
    {
      LC <- tree$node[[i]]$left
      RC <- tree$node[[i]]$right

      #check to see if there are no observations inside both the children
      if(is.null(tree$node[[LC]]$obs) || is.null(tree$node[[RC]]$obs))
      {
        #delete both children if there are no obs in either child node
        tree$node[[i]]$left <- NULL
        tree$node[[i]]$right <- NULL
      }
    }
  }

  return(tree)
}
```

7.7 treeCopy Code

This function creates a fresh copy without references to the deleted nodes from nodeChop.

```
treeCopy <- function(tree)
{
  #case where the root splits
  #as of nodeChop, the root always splits.
  if(is.null(tree$node[[tree$root]]$left) == FALSE)
  {

    #initialize the list
    newTree <- list(node=list(), fitness=0.0, root=1)

    #create a "placeholder" for future values.
    nullNode <- list(parent=NULL, left=NULL, right=NULL, splitVar=NULL, splitType=NULL,
    splitVal=NULL, model=NULL, depth = NULL, obs=NULL, obsLength = NULL )

    #counter for the node that is being written in the new tree.
    n <- 1

    #create a "placeholder" for future values.
    newTree$node[[n]] <- nullNode

    #copy over the root
    newTree$node[[n]] <- tree$node[[tree$root]]

    #note that the children may have non-sequential indexes.

    #counter for the last available node in the list.
    #this is a reference to the nodes in the new tree.
    lastNode <- 1

    #write in the next available index in for the left child because the root is split.
    newTree$node[[n]]$left <- lastNode + 1

    #write in the next available index in for the right child
    newTree$node[[n]]$right <- lastNode + 2

    #write in the placeholders for the left child of the root
    newTree$node[[lastNode + 1]] <- nullNode

    #write in the placeholders for the right child of the root
    newTree$node[[lastNode + 2]] <- nullNode

    #abbreviate old tree node numbers for simplicity
    LC.old <- tree$node[[tree$root]]$left
    RC.old <- tree$node[[tree$root]]$right

    #abbreviate new tree node numbers for simplicity
    LC.new <- newTree$node[[n]]$left
    RC.new <- newTree$node[[n]]$right

    #write in the old tree's node values.
    newTree$node[[LC.new]] <- tree$node[[LC.old]]
    newTree$node[[RC.new]] <- tree$node[[RC.old]]

    #write in the depth of the left child
    newTree$node[[LC.new]]$depth <- newTree$node[[n]]$depth + 1
```

```
#write in the depth of the right child
newTree$node[[RC.new]]$depth <- newTree$node[[n]]$depth + 1

#note that the child references of node 2,3 are going to be wrong at this point.

#parent will be okay because the root starts at node 1 (For now)

#add child nodes to list of nodes queued to be written
#but these are a reference to the new tree numbers?
nodesToCopy <- c((lastNode + 1):(lastNode + 2))

#update lastNode number
lastNode <- lastNode + 2

#while there are still nodes that need to be processed (Written over):
while(length(nodesToCopy > 0))
{
  # select top node from list, write the index into n. This is the node currently
  being examined.

  n <- nodesToCopy[1]

  #remove the first node in the queue
  nodesToCopy <- nodesToCopy[-1]

  #are there children? Note that the child numbers will be wrong at this point, b/c
  they reference the old tree!

  if(!is.null(newTree$node[[n]]$left))
  {
    #store the old child number info:
    LC.old <- newTree$node[[n]]$left
    RC.old <- newTree$node[[n]]$right

    #overwrite in the next available index in for the left child
    newTree$node[[n]]$left <- lastNode + 1

    #overwrite in the next available index in for the right child
    newTree$node[[n]]$right <- lastNode + 2

    #write in the placeholders for the left child
    newTree$node[[lastNode + 1]] <- nullNode

    #write in the placeholders for the right child
    newTree$node[[lastNode + 2]] <- nullNode

    #is this the group that needs deleting???
    #abbreviate new tree node numbers for simplicity
    LC.new <- newTree$node[[n]]$left
    RC.new <- newTree$node[[n]]$right

    #write in the old tree's node values.
    newTree$node[[LC.new]] <- tree$node[[LC.old]]
    newTree$node[[RC.new]] <- tree$node[[RC.old]]

    #but has child references in the old tree still!

    #update the parent references
    newTree$node[[LC.new]]$parent <- n
    newTree$node[[RC.new]]$parent <- n

    #write in the depth of the left child
```

```
      newTree$node[[LC.new]]$depth <- newTree$node[[n]]$depth + 1

      #write in the depth of the right child
      newTree$node[[RC.new]]$depth <- newTree$node[[n]]$depth + 1

      #add child nodes to list of nodes queued to copy
      nodesToCopy <- c(nodesToCopy, (lastNode+1):(lastNode+2))

      #increment nodesToCopy:
      lastNode <- lastNode + 2
    }

    #if there are no children in the current node, then it's going to be skipped.
  }

 }

 else
 {
  #initialize the list
  newTree <- list(node=list(), fitness=0.0, root=1)

  #create a "placeholder" for future values.
  nullNode <- list(parent=NULL, left=NULL, right=NULL, splitVar=NULL,splitType=NULL,
  splitVal=NULL, model=NULL, depth = NULL, obs=NULL, obsLength = NULL )

  newTree$node[[1]] <- nullNode

  #copy over the root
  newTree$node[[1]] <- tree$node[[tree$root]]
 }

 #new cleanup section to delete out the split rules, etc. from terminal nodes
 for(i in 1:length(newTree$node))
 {
  #if there are no children
  if(is.null(newTree$node[[i]]$left))
  {
   newTree$node[[i]]$splitVar <- NULL
   newTree$node[[i]]$splitType <- NULL
   newTree$node[[i]]$splitVal <- NULL
  }
 }

 return(newTree)
}
```

7.8 linearRegression Code

This function creates regression models in the terminal nodes.

```
linearRegression <- function(X, Y, Tree, simpleOrMultiple)
{
 #initialize a measure of the tree's SSE. This will be a simple sum for now.
 Tree$TotalSSE <- 0

 #call to loop over the whole tree

 #requires that there be no loose ends, the tree is nice and clean.
 #loop to go over all the nodes in the tree
```

```
for(i in 1:length(Tree$node))
{
  #if it's a terminal node (no left child)
  if(is.null(Tree$node[[i]]$left))
  {
    #create a vector for the observation #s in the node
    obsVector <- Tree$node[[i]]$obs

    #copy the predictor & Response values from the selected observations in the term.
    node.

    #the actual data will be in the vector and matrix
    LRRespData <- Y[obsVector,]
    LRPredData <- X[obsVector,]

    #simple linear regression
    if(simpleOrMultiple == 0)
    {
      #run a simple linear regression w/all variables, one at a time
      for(j in 1:ncol(LRPredData))
      {
        #linear regression call if the variable is a factor
        if(is.factor(LRPredData[[j]]))
        {
          #only run it if there is more than 1 level (o/w it won't run correctly)
          if(length(unique(LRPredData[[j]])) > 1)
          {
            #call for simple linear regression
            LRTemp <- lm(LRRespData ~ factor(LRPredData[[j]]))
            #the model is now in LRTemp

            #if there is no SSE in the node (first model is being fit currently)
            if(is.null(Tree$node[[i]]$SSE))
            {
              #write in the model into the node
              Tree$node[[i]]$model <- LRTemp
              #write in the variable used as the "best variable"
              Tree$node[[i]]$bestVar <- j
              #write in the SSE into the node:
              Tree$node[[i]]$SSE <- sum((LRTemp$residuals)^2)
            }

            #on subsequent models, if the SSE beats the current best SSE:
            if(sum((LRTemp$residuals)^2) < Tree$node[[i]]$SSE)
            {
              #write in the model into the node
              Tree$node[[i]]$model <- LRTemp
              #write in the variable used as the "best variable"
              Tree$node[[i]]$bestVar <- j
              #write in the SSE into the node:
              Tree$node[[i]]$SSE <- sum((LRTemp$residuals)^2)
            }
          }


        }

        #linear regression call if the variable is not a factor (Quant)
        else
        {
          #note that the dep var should be in the same data set as the ind vars.
          LRTemp <- lm(LRRespData ~ LRPredData[[j]])
```

```
      #if there is no SSE in the node (first model is being fit currently)
      if(is.null(Tree$node[[i]]$SSE))
      {
        #write in the model into the node
        Tree$node[[i]]$model <- LRTemp
        #write in the variable used as the "best variable"
        Tree$node[[i]]$bestVar <- j
        #write in the SSE into the node:
        Tree$node[[i]]$SSE <- sum((LRTemp$residuals)^2)
      }

      #on subsequent models, if the deviance beats the current best deviance:
      if(sum((LRTemp$residuals)^2) < Tree$node[[i]]$SSE)
      {
        #write in the model into the node
        Tree$node[[i]]$model <- LRTemp
        #write in the variable used as the "best variable"
        Tree$node[[i]]$bestVar <- j
        #write in the SSE into the node:
        Tree$node[[i]]$SSE <- sum((LRTemp$residuals)^2)
      }
    }


  }

}

#stepwise multiple regression
else
{
  badVariables <- NULL

  #check to see if all are alright:
  for(j in 1:ncol(LRPredData))
  {
    #linear regression call if the variable is a factor
    if(is.factor(LRPredData[[j]]))
    {
      #only run it if there is more than 1 level (o/w it won't run correctly)
      if(length(unique(LRPredData[[j]])) == 1)
      {
        badVariables <- c(badVariables, j)

      }
    }
  }

  if(!is.null(badVariables))
  {
    #delete out the bad column of data for usage
    LRPredData <- LRPredData[,-badVariables]
  }

  #full multiple regression
  LRTemp <- lm(LRRespData ~ ., data = LRPredData )
  Tree$node[[i]]$model <- LRTemp

  #stepwise regression with BIC
  #Tree$node[[i]]$model <- step(LRTemp, direction = "backward",
  k=log(nrow(LRPredData)), trace=0)
```

```
      #get the SSE from the stepwise regression
      Tree$node[[i]]$SSE <- sum((Tree$node[[i]]$model$residuals)^2)

      Tree$node[[i]]$bestVar <- "Multiple"
    }

    #write in the deviance for the final model into the tree's fitness value
    Tree$TotalSSE <- Tree$TotalSSE + Tree$node[[i]]$SSE

    #write in the deviance for the final model into the tree's fitness value
    Tree$fitness <- 1/(Tree$TotalSSE)
  }
 }

 n <- length(X[,1])

 #First BIC = # of parameters in the terminal nodes + 1 for the constant variance
 across the TNs.

 #initialize first BIC (for the constant variance)
 p_BIC_1 <- 1

 for(k in 1:length(Tree$node))
 {
  p_BIC_1 <- p_BIC_1 + length(Tree$node[[k]]$model$coef)
 }

 #second proposed BIC is the first with the addition of the # of splits.
 p_BIC_2 <- p_BIC_1 + (length(Tree$node) - 1)/2

 Tree$BIC_1 <- (-n/2)*log(2*(pi), base = exp(1)) - (n/2)*log((Tree$TotalSSE)/n, base
 = exp(1)) - (n/2) - 0.5 * p_BIC_1 * log(n, base = exp(1))

 Tree$BIC_2 <- (-n/2)*log(2*(pi), base = exp(1)) - (n/2)*log((Tree$TotalSSE)/n, base
 = exp(1)) - (n/2) - 0.5 * p_BIC_2 * log(n, base = exp(1))

 Tree$AIC_1 <- (-n/2)*log(2*(pi), base = exp(1)) - (n/2)*log((Tree$TotalSSE)/n, base
 = exp(1)) - (n/2) - p_BIC_1

 Tree$AIC_2 <- (-n/2)*log(2*(pi), base = exp(1)) - (n/2)*log((Tree$TotalSSE)/n, base
 = exp(1)) - (n/2) - p_BIC_2

 return(Tree)

}
```

7.9 treeTableLinear Code

This function creates a table for a decision tree.

```
treeTableLinear <- function(Tree)
{
 #initialize the columns.
 node.number <- NULL
 parent <- NULL
 left.child <- NULL
 right.child <- NULL
 split.type <- NULL
 split.variable <- NULL
```

```
split.value <- NULL
number.categories <- NULL
number.obs <- NULL
terminal.node <- NULL
best.linear.variable <- NULL
node.sse <- NULL
depth <- NULL
total.sse <- NULL
MSE <- NULL
BIC_1 <- NULL
BIC_2 <- NULL
AIC_1 <- NULL
AIC_2 <- NULL

#loop to cover all nodes:
for(i in 1:length(Tree$node))
{
  #write in the node number for the row.
  node.number[i] <- i

  if(is.null(Tree$node[[i]]$parent[i]) == FALSE)
  {
    parent[i] <- Tree$node[[i]]$parent
  }

  if(is.null(Tree$node[[i]]$parent[i]) == TRUE)
  {
    parent[i] <- NA
  }

  #The number of obs used to be here. If it was, then it would NA out the rest of the
  columns for some reason.

  if(is.null(Tree$node[[i]]$left) == FALSE)
  {
    left.child[i] <- Tree$node[[i]]$left
  }

  if(is.null(Tree$node[[i]]$left) == TRUE)
  {
    left.child[i] <- NA
  }

  if(is.null(Tree$node[[i]]$right) == FALSE)
  {
    right.child[i] <- Tree$node[[i]]$right
  }

  if(is.null(Tree$node[[i]]$right) == TRUE)
  {
    right.child[i] <- NA
  }

  if(is.null(Tree$node[[i]]$splitType) == FALSE)
  {
    #if it's a categorical variable
    if(Tree$node[[i]]$splitType)
    {
      split.type[i] <- "Cat"
    }

    #if it's not a categorical variable
    else
```

```
  {
    split.type[i] <- "Quant"
  }
}

if(is.null(Tree$node[[i]]$splitType) == TRUE)
{
  split.type[i] <- NA
}

if(is.null(Tree$node[[i]]$splitVar) == FALSE)
{
  split.variable[i] <- paste(Tree$node[[i]]$splitVar, collapse = " ")
}

if(is.null(Tree$node[[i]]$splitVar) == TRUE)
{
  split.variable[i] <- NA
}

if(is.null(Tree$node[[i]]$splitVal) == FALSE)
{
  number.categories[i] <- length(Tree$node[[i]]$splitVal)

  split.value[i] <- paste(Tree$node[[i]]$splitVal, collapse = " ")

}

if(is.null(Tree$node[[i]]$splitVal) == TRUE)
{
  number.categories[i] <- NA
  split.value[i] <- NA
}

if(is.null(Tree$node[[i]]$bestVar) == FALSE)
{
    best.linear.variable[i] <- Tree$node[[i]]$bestVar
}

if(is.null(Tree$node[[i]]$bestVar) == TRUE)
{
    best.linear.variable[i] <- NA
}

if(is.null(Tree$node[[i]]$SSE) == FALSE)
{
  node.sse[i] <- Tree$node[[i]]$SSE

  if(is.null(Tree$node[[i]]$obsLength) == FALSE)
  {
    MSE[i] <- node.sse[i]/Tree$node[[i]]$obsLength
  }
}

if(is.null(Tree$node[[i]]$SSE) == TRUE)
{
  node.sse[i] <- NA
}



if(is.null(Tree$node[[i]]$depth) == FALSE)
{
```

```
    depth[i] <- Tree$node[[i]]$depth
   }

   if(is.null(Tree$node[[i]]$depth) == TRUE)
   {
    depth[i] <- NA
   }

   #if it is a terminal node:
   if((is.null(Tree$node[[i]]$left) == TRUE) && (is.null(Tree$node[[i]]$right) ==
   TRUE))

   {
    terminal.node[i] <- TRUE
   }

   if((is.null(Tree$node[[i]]$left) == FALSE) || (is.null(Tree$node[[i]]$right) ==
   FALSE))

   {
    terminal.node[i] <- NA
   }

   total.sse[i] <- Tree$TotalSSE

   BIC_1[[i]] <- Tree$BIC_1

   BIC_2[[i]] <- Tree$BIC_2

   AIC_1[[i]] <- Tree$AIC_1

   AIC_2[[i]] <- Tree$AIC_2

   #The following two used to be above where the comment is. It seems to work now.

   if(is.null(Tree$node[[i]]$obsLength) == FALSE)
   {
    number.obs[i] <- Tree$node[[i]]$obsLength
   }

   if(is.null(Tree$node[[i]]$obsLength) == TRUE)
   {
    number.obs[i] <- NA
   }
  }

  #combine everything into a data frame:
  tree.Table <- data.frame(node.number, parent, left.child, right.child, depth,
  split.type, split.variable, number.categories, split.value, number.obs,
  terminal.node, best.linear.variable, node.sse, MSE, total.sse, BIC_1, BIC_2, AIC_1,
  AIC_2)

  #return the data frame:
  return(tree.Table)
}
```

7.10 forestTableLinear Code

This function creates a table outlining the fitness measures and characteristics of the trees in a forest.

```
forestTableLinear <- function(Forest)
{
  #initialize the columns.
  tree.number <- NULL
  num.nodes <- NULL
  tree.sse <- NULL
  tree.fitness <- NULL
  source.tree <- NULL
  mutated.node <- NULL
  change <- NULL
  maxDepth <- NULL
  BIC_1 <- NULL
  BIC_2 <- NULL
  AIC_1 <- NULL
  AIC_2 <- NULL

  #loop to cover all Trees:
  #this accounts for the counts of trees inserted at the end.

    for(i in 1:(length(Forest) - 1))
    {
      #write in the node number for the row.
      tree.number[i] <- i

      #write in the number of nodes the tree has
      num.nodes[i] <- length(Forest[[i]]$node)

      if(!is.null(Forest[[i]]$TotalSSE))
      {
        tree.sse[i] <- Forest[[i]]$TotalSSE
      }

      if(is.null(Forest[[i]]$TotalSSE))
      {
        tree.sse[i] <- NA
      }

      if(!is.null(Forest[[i]]$fitness))
      {
        tree.fitness[i] <- Forest[[i]]$fitness
      }

      if(!is.null(Forest[[i]]$sourceTree))
      {
        source.tree[i] <- paste(Forest[[i]]$sourceTree, collapse = ",")
      }

      if(is.null(Forest[[i]]$sourceTree))
      {
        source.tree[i] <- NA
      }

      if(!is.null(Forest[[i]]$mutatedNode))
      {
        mutated.node[i] <- paste(Forest[[i]]$mutatedNode, collapse = ",")
```

```
}


if(is.null(Forest[[i]]$mutatedNode))
{
  mutated.node[i] <- NA
}

if(!is.null(Forest[[i]]$mutationType))
{
  change[i] <- Forest[[i]]$mutationType
}

if(is.null(Forest[[i]]$mutationType))
{
  change[i] <- NA
}

if(!is.null(Forest[[i]]$BIC_1))
{
  BIC_1[[i]] <- Forest[[i]]$BIC_1
}

if(is.null(Forest[[i]]$BIC_1))
{
  BIC_1[[i]] <- NA
}

if(!is.null(Forest[[i]]$BIC_2))
{
  BIC_2[[i]] <- Forest[[i]]$BIC_2
}

if(is.null(Forest[[i]]$BIC_2))
{
  BIC_2[[i]] <- NA
}

if(!is.null(Forest[[i]]$AIC_1))
{
  AIC_1[[i]] <- Forest[[i]]$AIC_1
}

if(is.null(Forest[[i]]$AIC_1))
{
  AIC_1[[i]] <- NA
}

if(!is.null(Forest[[i]]$AIC_2))
{
  AIC_2[[i]] <- Forest[[i]]$AIC_2
}

if(is.null(Forest[[i]]$AIC_2))
{
  AIC_2[[i]] <- NA
}

if(length(Forest[[i]]$node) > 0)
{
  maxDepth[i] <- Forest[[i]]$node[[1]]$depth

  if(length(Forest[[i]]$node) >= 2)
```

```
      {
        #loop over all the nodes
        for(j in 2:length(Forest[[i]]$node))
        {
          if(Forest[[i]]$node[[j]]$depth > maxDepth[i])
          {
            maxDepth[i] <- Forest[[i]]$node[[j]]$depth
          }
        }
      }
    }

    else
    {
      maxDepth[i] <- NA
    }
  }


  #combine everything into a data frame:
  forest.Table <- data.frame(tree.number, num.nodes, maxDepth, tree.sse, tree.fitness,
  source.tree, mutated.node, change, BIC_1, BIC_2, AIC_1, AIC_2)

  #return the data frame:
  return(forest.Table)

#end of the function
}
```

7.11 testLinear Code

This function partitions and evaluates test data through the best performing trees found per terminal node size. The *BIC* training values are retained in order to determine which of the trees is chosen as the champion model.

```
testLinear <- function(X, Y, bestTrees, simpleOrMultiple, obliqueSplits)
{
  #loop over all the best trees by number of terminal nodes
  #recall the last one is a count of trees!!!!
  for(m in 1:(length(bestTrees) - 1))
  {
    #if there actually is a tree there:
    if(bestTrees[[m]]$fitness != 0)
    {
      #initialize a placeholder for the SSE from the new obs:
      fittedValues <- NULL
      errors <- NULL

      #copy bestTrees[[m]] to a temp tree to agree with previous plinko code:
      Tree <- bestTrees[[m]]

      #initialize the testSSE for a new tree
      Tree$TotalSSE <- 0

      #loop over all the nodes in the mth best tree
      for(p in 1:length(Tree$node))
```

```
{
  #delete out the observations to rewrite them
  Tree$node[[p]]$obs <- NULL
  Tree$node[[p]]$obsLength <- NULL
  Tree$node[[p]]$SSE <- NULL
}


#plinko the new data:

#write in the list of all the observations into the first node's observations
Tree$node[[1]]$obs <- c(1:nrow(X))

# add root node to list of nodes to process
nodesToProcess <- c(Tree$root)

#begin loop to sort
#while there are still nodes down the path that haven't been sorted yet
while(length(nodesToProcess) > 0)
{
  #write in the first node index to process into i.
  i <- nodesToProcess[1]

  #check to see if there are obs to divide up
  if(!is.null(Tree$node[[i]]$obs))
  {
    # check to see if it's NOT a terminal node (has children)
    if(!is.null(Tree$node[[i]]$left))
    {
      #add the child nodes to split (divide) later.
      nodesToProcess <- c(nodesToProcess, Tree$node[[i]]$left,
      Tree$node[[i]]$right)

      #remove the top node from the queue
      nodesToProcess <- nodesToProcess[-1]

      # if is a categorical variable and factor split
      if(Tree$node[[i]]$splitType == TRUE)
      {
        #write in the left child node index into j for ease of reading
        j <- Tree$node[[i]]$left

        #write in the right child node index into k for ease of reading
        k <- Tree$node[[i]]$right

        #write in the obs that are both in the current node
        #and a subset of those that are in the split set
        obs.for.left <- intersect(Tree$node[[i]]$obs,
        which(is.element(X[,Tree$node[[i]]$splitVar],Tree$node[[i]]$splitVal)))

        #write in the obs that are both in the current node and NOT in the left
        Child

        obs.for.right <- setdiff(Tree$node[[i]]$obs, obs.for.left)

        #write in the nodes:
        Tree$node[[j]]$obs <- obs.for.left
        Tree$node[[k]]$obs <- obs.for.right
      }

      # quant split
      else
      {
```

```
#if no oblique splits:
if(length(Tree$node[[i]]$splitVal) == 1)
{
  #write in the left child node index into j for ease of reading
  j <- Tree$node[[i]]$left

  #write in the right child node index into k for ease of reading
  k <- Tree$node[[i]]$right

  #write in the obs that are both in the current node and less than or
  equal to the split value

  obs.for.left <- intersect(Tree$node[[i]]$obs,
  which(X[,Tree$node[[i]]$splitVar] <= Tree$node[[i]]$splitVal))

  if(length(obs.for.left) == 0)
  {
    obs.for.left <- NULL  # is this how to handle this situation?
  }

  #write in the obs that are both in the current node and NOT in the left
  Child

  obs.for.right <- setdiff(Tree$node[[i]]$obs, obs.for.left)

  #write in the nodes:
  Tree$node[[j]]$obs <- obs.for.left
  Tree$node[[k]]$obs <- obs.for.right
}

#if there are oblique splits:
else
{
  #write in the left child node index into j for ease of reading
  j <- Tree$node[[i]]$left

  #write in the right child node index into k for ease of reading
  k <- Tree$node[[i]]$right


  #write in the obs that are both in the current node and less than or
  equal to the split value

  slope <- (Tree$node[[i]]$splitVal[2] –
  Tree$node[[i]]$splitVal[1])/(Tree$node[[i]]$splitVal[4] –
  Tree$node[[i]]$splitVal[3])

  RHS <- (slope * (X[,Tree$node[[i]]$splitVar[2]] -
  Tree$node[[i]]$splitVal[3])) + Tree$node[[i]]$splitVal[1]

  obs.for.left <- intersect(Tree$node[[i]]$obs,which(RHS <=
  X[,Tree$node[[i]]$splitVar[1]]) )

  if(length(obs.for.left) == 0)
  {
    obs.for.left <- NULL  # is this how to handle this situation?
  }

  #write in the obs that are both in the current node and NOT in the left
  Child

  obs.for.right <- setdiff(Tree$node[[i]]$obs, obs.for.left)
```

```
        #write in the nodes:
        Tree$node[[j]]$obs <- obs.for.left
        Tree$node[[k]]$obs <- obs.for.right
      }
    }

    #I don't think this is necessary:
    #if nothing goes one way or the other, delete the obs list in the empty
    Nodes

    if(length(Tree$node[[j]]$obs) == 0)
    {
      Tree$node[[j]]$obs <- NULL
    }

    if(length(Tree$node[[k]]$obs) == 0)
    {
      Tree$node[[k]]$obs <- NULL
    }
  }

  # if it is a terminal node
  else
  {
    #remove the top node from the queue
    nodesToProcess <- nodesToProcess[-1]
  }
}

#if there are no obs to divide up
else
{
  #remove the top node from the queue
  nodesToProcess <- nodesToProcess[-1]
}

#end of the loop over all the nodes to process
}

#input the number of obs per node
for(i in 1:length(Tree$node))
{
  #check to make sure that it's not empty:
  if(!is.null(Tree$node[[i]]$obs))
  {
    Tree$node[[i]]$obsLength <- length(Tree$node[[i]]$obs)
  }
}

#at this point, all the obs should be plinko'd into the appropriate terminal
nodes.

#need to run the model and save the Errors.


#for simple linear regression:
if(simpleOrMultiple == 0)
{
  #loop over all the mth tree's nodes:
  for(i in 1:length(Tree$node))
  {
    #if the node is terminal:
    if(is.null(Tree$node[[i]]$left))
```

```
   {
     #if there are obs in the terminal node:
     if(!is.null(Tree$node[[i]]$obs))
     {
       #loop over all the observations:
       for(j in 1:Tree$node[[i]]$obsLength)
       {
         #if it's a cat variable:
         if(is.factor(X[,Tree$node[[i]]$bestVar]))
         {
           #if there is a coefficient returned for the variable value:
           if(!is.na(Tree$node[[i]]$model$coefficients[
           paste("factor(LRPredData[[j]])" , X[Tree$node[[i]]$obs[j],
           Tree$node[[i]]$bestVar], sep="")]))
           {
             fittedValues[j] <- sum(Tree$node[[i]]$model$coefficients[1] +
             Tree$node[[i]]$model$coefficients[paste("factor(LRPredData[[j]])",
             X[Tree$node[[i]]$obs[j], Tree$node[[i]]$bestVar], sep="")])

             errors[j] <- Y[Tree$node[[i]]$obs[j],1] - fittedValues[j]
           }

           else
           {
             fittedValues[j] <- sum(Tree$node[[i]]$model$coefficients[1])
             errors[j] <- Y[Tree$node[[i]]$obs[j],1] - fittedValues[j]
           }
         }

         #if it's a quantitative variable:
         else
         {
           fittedValues[j] <- Tree$node[[i]]$model$coefficients[1] +
           (Tree$node[[i]]$model$coefficients[2] * X[Tree$node[[i]]$obs[j],
           Tree$node[[i]]$bestVar])

           #need to calculate the errors:
           errors[j] <- Y[Tree$node[[i]]$obs[j],1] - fittedValues[j]
         }
       }

       #update the SSE
       Tree$node[[i]]$SSE <- sum(errors^2)

       #update the total SSE:
       Tree$TotalSSE <- Tree$TotalSSE + Tree$node[[i]]$SSE

       #update the fitness
       Tree$fitness <- 1/Tree$TotalSSE
     }

   }

   #clear the fitted values and errors
   fittedValues <- NULL
   errors <- NULL

  }
}

#multiple regression case:
if(simpleOrMultiple == 1)
{
```

```
      #loop over all the mth tree's nodes:
      for(i in 1:length(Tree$node))
      {
        #if the node is terminal:
        if(is.null(Tree$node[[i]]$left))
        {
          #if there are obs in the terminal node:
          if(!is.null(Tree$node[[i]]$obs))
          {
            #create a vector for the observation #s in the node
            obsVector <- Tree$node[[i]]$obs

            #copy the predictor & Response values from the selected observations in
            the term. node.
            #the actual data will be in the vector and matrix
            LRRespData <- Y[obsVector,]
            LRPredData <- X[obsVector,]

            #create a vector for the sse
            errors <- LRRespData - predict(Tree$node[[i]]$model, newdata = LRPredData)


            #update the SSE
            Tree$node[[i]]$SSE <- sum(errors^2)

            #update the total SSE:
            Tree$TotalSSE <- Tree$TotalSSE + Tree$node[[i]]$SSE

            #update the fitness
            Tree$fitness <- 1/Tree$TotalSSE
          }

        }

        #clear the fitted values and errors
        errors <- NULL
      }
    }


    #rewrite the tree into the bestTrees
    bestTrees[[m]] <- Tree

  #end of loop to make sure that there is a tree to process. i.e. missing a tree with
  a certain number of Terminal Nodes.

  }

#end of for loop over all the m trees
}

#return the TotalSSE by number of terminal nodes in the trees.
return(bestTrees)

#end of function
}
```