LINK DISCOVERY IN VERY LARGE GRAPHS BY CONSTRUCTIVE INDUCTION USING GENETIC PROGRAMMING

by

TIMOTHY EDWARDS WENINGER

B.S., Kansas State University, 2007

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2008

Approved by:

Major Professor
William H. Hsu

Abstract

This thesis discusses the background and methodologies necessary for constructing features in order to discover hidden links in relational data. Specifically, we consider the problems of predicting, classifying and annotating friends relations in friends networks, based upon features constructed from network structure and user profile data. I first document a data model for the blog service *LiveJournal*, and define a set of machine learning problems such as predicting existing links and estimating inter-pair distance. Next, I explain how the problem of classifying a user pair in a social networks, as directly connected or not, poses the problem of selecting and constructing relevant features. In order to construct these features, a genetic programming approach is used to construct multiple symbol trees with base features as their leaves; in this manner, the genetic program selects and constructs features that many not have been considered, but possess better predictive properties than the base features. In order to extract certain graph features from the relatively large social network, a new shortest path search algorithm is presented which computes and operates on a Euclidean embedding of the network. Finally, I present classification results and discuss the properties of the frequently constructed features in order to gain insight on hidden relations that exists in this domain.

Table of Contents

List of Figures vi

List of Tables viii

Acknowledgements ix

CHAPTER 1 - Introduction 1

1.1 Inductive Learning 3

1.1.1 Feature Extraction 6

1.1.2 Selective Induction 8

1.1.3 Constructive Induction 10

1.2 Evolutionary Computation 14

1.2.1 Traditional Search Methods and Local Maxima 15

1.2.2 Genetic Algorithms 17

1.2.2.1 Genetic Operations 18

1.2.2.2 Probabilistic Selection 21

1.2.3 Genetic Programming 23

1.2.3.1 Representation 24

1.2.3.2 Genetic Operations 25

1.3 Graph Theory 28

1.3.1 Directed Graphs 29

1.3.2 Social Networks 30

1.3.3 Link Mining 32

1.4 Principal Claims 33

CHAPTER 2 - Related Research 35

2.1 Meta Learning 35

2.1.1 Bagging 36

2.1.2 Boosting 37

2.1.3 Random Forests 37

2.2 Krawiec's Feature Construction with Genetic Programs Approach 38

2.3 Hsu's Feature Selection for Link Mining Approach 39

CHAPTER 3 - Methodology 40
3.1 Crawling a social network 40
3.1.1 Crawl Results 43
3.2 Base Feature Selection 44
3.2.1 User-dependent Base Features 44
3.2.2 Pair-dependent Base Features 44
3.2.3 Graph Base Features 45
3.3 Candidate Pair Generation 45
3.3.1 Graph Density 46
3.3.2 Forced Parity Candidate Generation 46
3.4 Genetic Feature Construction 47
3.4.1 Symbolic Regression 47
3.4.2 Feature Construction via Multiple Regression Trees 48
3.5 Feature Computation 49
3.5.1 Shortest Path Approximation 50
3.5.1.1 Graph Embedding 51
3.5.1.2 Finding the Approximate Shortest Path 51
3.6 Experiment Design 52
3.6.1 Evaluation Metrics 53
CHAPTER 4 - Results 55
4.1 Results from Traditional Learning using Base Features 55
4.2 Results of Feature Construction using a Single Symbol Tree 60
4.3 Results of Feature Construction using Multiple Symbol Trees 61
4.3.1 OneR 61
4.3.2 Logistic 63
4.3.3 J48 64
4.3.4 NaiveBayes 66
4.3.5 IB1 67
4.3.6 Comparative Analysis 69
4.3.7 Constructed Feature Trees 70
4.4 Meta Learning Results 72

CHAPTER 5 - Conclusions and Future Work 74
5.1 Large Graph Crawling and Feature Extraction 75
5.2 Efficient Feature Computation on Large Graphs 76
5.3 Analysis of Genetically-Constructed Features 77
5.4 Prediction Performance of Genetically-Constructed Features 78
5.5 Future Work 80
Bibliography 81
Appendix A - Complete Feature Set 87
Appendix B - ECJ Parameters File 89
Appendix C - Symbol Trees for Best Individuals 96

List of Figures

Figure 1.1: An example decision tree for $A \vee B \wedge C \vee D$ 12
Figure 1.2: Decision tree from Figure 1.1 with two newly constructed features.......... 12
Figure 1.3: Example search space gradient with two local maxima (A, B) and a global optimum (C). 16
Figure 1.4: Example of reproduction when applied on bit-sequence 100101. 19
Figure 1.5: Example of mutation occurring on bit #3 in the bit-sequence 100101. 20
Figure 1.6: Example crossover of the bit-sequences 100100 and 001111. 21
Figure 1.7: Roulette wheel visualization for normalized fitness 23
Figure 1.8: Pseudocode for a genetic algorithm. 23
Figure 1.9: Tree representation of the equation $Z - 4X + YW + 9$ for an individual in a genetic program. 25
Figure 1.10: Example of GP mutation where the individual from Figure 1.9 is mutated to form a new individual. 26
Figure 1.11: Example of GP crossover where two parent individuals are split and recombined to form two new individuals. 27
Figure 1.12: Example undirected graph with five vertices and five edges. 28
Figure 1.13: Example digraph with five vertices and six arcs. 29
Figure 2.1: Taken with permission from the feature construction Genetic programming-based framework for [Kra02]. 38
Figure 3.1: Pseudocode for dynamic thread control in LJCrawler 41
Figure 3.2: Storage schema for MySQL database 43
Figure 3.3: Example symbol tree used in Table 3.1 48
Figure 4.1: Decision tree for J48 classifier trained on base features. 58
Figure 4.2: ROC curves for learning algorithms trained and cross-validated 59
Figure 4.3: Progression of the fitness of the best individual through 4 generations. Higher is better. 60
Figure 4.4: Best individuals per generation with OneR for 5 GP runs. Lower is better. 62
Figure 4.5: Average validation score over the entire population per generation with OneR for 5 GP runs. Lower is better. 62

Figure 4.6: Best individuals per generation with Logistic for 5 GP runs. Lower is better. 63
Figure 4.7: Average validation score over the entire population per generation with Logistic for 5 GP runs. Lower is better. 64
Figure 4.8: Best individuals per generation with J48 for 5 GP runs. Lower is better. 65
Figure 4.9: Average validation score over the entire population per generation with J48 for 5 GP runs. Lower is better. 65
Figure 4.10: Best individuals per generation with NaiveBayes for 5 GP runs. Lower is better... 66
Figure 4.11: Average validation score over the entire population per generation with NaiveBayes for 5 GP runs. Lower is better. 67
Figure 4.12: Best individuals per generation with IB1 for 4 GP runs. Lower is better. 68
Figure 4.13: Average validation score over the entire population per generation with IB1 for 4 GP runs. Lower is better. 68
Figure 4.14: Average validation scores for the best individuals for all generations. Lower is better. 69
Figure 4.15: An example genetically-constructed condensed symbol tree. 70
Figure 4.16: An example genetically constructed symbol tree with 71
Figure 4.17: ROC curves for learning algorithms trained and cross validated on feature data set. 73
Figure 5.1: Comparison of results from classifiers trained by (1) genetically-constructed features (2) base features and (3) meta-learning algorithms. 79

List of Tables

Table 1.1: Example problem individuals with fitness values and corresponding selection probabilities 22
Table 1.2: Types of link in the blog service *LiveJournal* 31
Table 3.1: Example Symbolic Regression Interpretation 48
Table 3.2: Example of features constructed with multiple regression trees 49
Table 4.1: Attribute Ranking in terms of 56
Table 4.2: Results for classifiers trained on base features, tested with 57
Table 4.3: Results for classifiers trained on base features, tested 59
Table 4.4: Symbol trees and corresponding fitness for 4 generations of single symbolic regression 61
Table 4.5: Classifier results in terms of AUC (1 – fitness) compared to base results. Higher is better. 70
Table 4.6: Top 10 features ordered by their prevalence in individuals' symbol trees 71
Table 4.7: Results for meta-learning algorithms trained on base features, ... 72

Acknowledgements

I would like to thank my wife, Jordan, for her patience and understanding during the formulation and writing of this thesis. I also thank my major professor, Dr. William Hsu, and my committee, including Dr. Dan Andresen and Dr. Doina Caragea, for their help and guidance throughout the construction of this work.

I would also like to thank the Defense Intelligence Agency and the National Agriculture Biosecurity Center for their sponsorship of this work, the Department of Homeland Security and the University of Illinois Urbana-Champaign for my fellowship at the Multimodal Information Access and Synthesis Summer Institute, Dr. Krzysztof Krawiec for permission to use his illustration in this thesis, and Dr. Rod Howell, Dr. Todd Easton, Dr. Caterina Scoglio, Dr. Janis Crow, and Dr. Jiawei Han for their useful discussions during the formulation of this thesis.

CHAPTER 1 - Introduction

Traditional data mining tasks such as association rule mining or market basket analysis attempt to find patterns in a dataset. According to Getoor, “This is consistent with the classical statistical inference problem of trying to identify a model given a random sample from a common underlying distribution” [Get03]. However, it is important to also data mine datasets that are relational, semi-structured or otherwise consist of links between various entities. These links can be explicit, such as an anchor tag in a web page, or implied such as a join operation in a relational database. As shown by the *PageRank* algorithm used by the popular search engine *Google*, link existence can be exploited to improve the predictive accuracy of learned models. [Bri98] Intuitively, attributes of linked objects are often more closely related than those of unlinked objects, and links are more likely to exist between objects that share common attributes. [Get03]

Feature construction in the multi-relational setting is also possible. Traditionally, the attributes of an object provide the basic description of the object. However, by leveraging the information contained in the relationships of objects, more information about the object can be gleaned providing the learning algorithm an appropriate context for a better induction model.

This thesis focuses mainly on link discovery, *i.e.*, predicting the existence of links between objects. In order to discover links not previously known to exist, a genetic programming approach is used to construct features that appropriately leverage the knowledge contained in the presence or absence of links. One computational challenge is that the graph that is studied is prohibitively large for traditional graph analysis algorithms to operate effectively. Therefore efficient memory-management techniques are devised to manage the underlying data structure.

This introductory chapter contains necessary information for readers who are unfamiliar with one or more of the following topics: constructive induction, genetic and evolutionary computation, social networks, and link mining, as well as a brief background study of the earlier work this thesis builds upon. This chapter is not intended to be a complete introduction to the topic; I refer the interested reader to Mitchell's book on machine learning [Mit97], Han and Kamber's book on data mining [Han06], Koza's book on genetic programming [Koz98], Wasserman and Faust's book on social networks [Was07], and Lise Getoor's paper on link mining [Get03]. In the final section of this chapter the principal claims of this thesis are stated. These claims are defended in the proceeding chapters.

Chapter 2 contains a brief overview of the related research. The first section describes an alternative approach called meta-learning while the next two sections on Krawiec's GP-based construction of features and Hsu's link mining approaches serve as motivations for this thesis.

Chapter 3 discusses the methodology of my approach. The first section describes how a large social network is crawled and locally stored as a graph. Subsequent sections describe the setup and execution of the experiments to be performed and the means by which results are gathered and presented.

Chapter 4 presents four sets of results for each of the four experiments described in Chapter 3. The standard results metrics will often-times be accompanied by a more descriptive analysis of the results. The final section of this chapter will present the results in a comparative context leading towards the final chapter and its coverage of conclusions that can be drawn from these experiments.

Chapter 5 contains discussion about and interpretations of the results by systematically demonstrating how well the experiments support the principal claims of this thesis. Finally, this chapter presents possibilities for future research and concludes the thesis.

1.1 Inductive Learning

Inductive learning, as an active research area of machine learning, explores algorithms that reason from external *examples* (also known as *instances* or *cases*) to produce general *theories* (or *hypotheses*), which then inductively make predictions about more examples. Examples, in this sense, are representations of concrete knowledge about individuals, including specific characteristics of the set of individuals, as well as an assessment of the individuals. For instance, the medical records and current symptoms of a patient is an *example* and the doctors' diagnosis of that patient is a *theory* because of the doctor's experience with previous patients. Externally supplied examples used for generating theories are typically known as *training examples*. Created theories should be more general then the specific examples from which they are derived because theories can make predictions about not only trained examples but also previously unseen examples. This generalization is at the core of inductive reasoning. Of course, there is no guarantee that the results induced by an inductive learner are completely correct even if all of the training examples are correct. Therefore, gauging the validity of a particular learning algorithm is often determined by the accuracy or error rates of the learned theories on previously unseen examples.

In the medical example provided above, if the training examples are provided with known labels such as the diagnoses of an illness for the patients then the inductive learning algorithm is called *supervised learning*. [Mit83] [Qui86] Supervised learning can solve two types of problems: *classification problems*, in which labels are categorical and *regression problems*, in which labels are continuous. [Wei95] This is in contrast to *unsupervised learning*, in which training examples are unlabeled or their labels are unknown. [Fis87] For classification problems, labels are referred to as *classes* and the induced theories are collectively called a *classifier*. In the likely instance that a problem involves only two classes, inductive learners typically consider examples to be either *positive* or *negative* examples of a *concept* [Qui86] [Mic83], in this case, the supervised learning task can be viewed as that of generating a definition of a concept.

Generally, these types of problems can be viewed as search problems [Mit82] involving a large *hypothesis space*, that is, the space consisting of all possible theories (hypotheses) under consideration. The goal of the search is to find the best theory with respect to the available training examples.

For each learning problem (*domain*), the set of attributes that is used to represent the examples is fixed and every example is represented using the same set of attributes. Moreover, all possible classes are fixed and mutually exclusive. A set of training examples is called the *training set*, and this set is usually a subset of all the examples in the domain. Examples that are used to test the learned theories are called *test examples* and they are typically different from the training examples. The most commonly used attributes are one of three types: *continuous* (*numeric*), nominal or binary. Continuous attributes have ordered values. An example of a continuous attribute is *age*. Nominal attributes have a fixed set of discrete values. Cardinal directions such as *north*, *south*, *east*, *west*, etc. are examples of nominal attributes. Binary attributes are a special case of nominal attributes where there are only two distinct values. An example of a binary attribute is a logical variable with values that can be either *true* or *false*.

Learned theories are often evaluated using two measurements: *generalization performance* (*accuracy*) and *theory complexity*. Generalization performance is the expected prediction accuracy of a theory when tested against unseen examples [Wol92]. To estimate the generalization performance prediction counts are made by tallying the four indicators:

1. *True Positives*: the number of correctly identified positively-labeled examples
2. *True Negatives*: the number of correctly identified negatively-labeled examples
3. *False Positives*: the number of incorrectly identified positively-labeled examples, *e.g.*, false alarm
4. *False Negatives*: the number of incorrectly identified negatively-labeled examples *e.g.*, miss

Furthermore, several generalization performance indicators are employed to determine the validity of a learned theory:

1. *Accuracy*: $(TP + TN)/(TP + TN + FP + FN)$, number correctly identified over total number.
2. *Precision*: $TP/(TP + FP)$, aka *Positive prediction value*
3. *Recall*: $TP/(TP + FN)$, aka *True positive rate*, *hit rate*, *sensitivity*
4. *F-Measure*: $2 * (Precision * Recall)/(Precision + Recall)$, *harmonic mean*
5. *Receiver Operating Characteristic* (ROC), trade off in true positive rate and false positive rate
6. *Area Under ROC,* aka *Area Under Curve* (AUC)

The theory complexity refers to the size of the description of the theory. For instance, a theory with many complex rules has a higher theory complexity than a theory with a single linear rule.

To learn a theory from a given set of training examples an induction algorithm needs to make some assumptions about the nature of theory that is being learned. In this context, these assumptions are called *biases*. The inclusion of a bias is necessary because the number of potential theories that are consistent with the training examples is usually prohibitively large. Furthermore, the training data does not necessarily indicate which theory is correct. [Mit80] In the same work, Mitchell comments that, "an unbiased learning system's ability to classify new instances is no better than if it simply stored all the training instances and performed a lookup when asked to classify a subsequent instance" [Mit80]. In other words, biases are used to reduce the hypothesis search space in order to make learning possible.

There are, essentially, two types of biases: *absolute* and *relative*. Absolute bias, otherwise known as *language bias*, assumes that the target theory belongs to some restricted set in the hypothesis space. This type of bias restricts the domain of theories that can be expressed and thus can be learned by the system because it is defined in terms of the description language of the learning system. For example, a decision tree model can only represent finite rules, whereas a neural network can represent a network of probabilities. Relative bias, otherwise known as *preference bias* or *search bias*, assumes that the theory to be learned is more likely to be from one set of hypothesis than from another. It places a preference ordering on hypothesis by directing the search through the hypothesis space. As an example, consider the following question: "How many marbles are in the jar?" To begin to formulate an answer to this question

one should consider all of the answers available. This set of potential answers is effectively a hypothesis space, and this hypothesis space would include meaningless answers, such as negative numbers, words that do not denote counting numbers, objects, or the plethora of conceivable hypothesis that would exist to answer any and all questions, ever. When venturing a guess at how many marbles are in the jar, valid answers can only be positive integers. Therefore, the bias applied to this example is that answers may only be positive integers. Learning algorithms that employ relative bias often cite Occam's Razor principle, which states that, "all other things being equal, the simplest solution is the best" [Blu87]. Occam's Razor is also known as *lex parsimoniae* or the law of parsimony. [Wik08a]

As stated above, every inductive learning algorithm uses some degree of bias. Therefore, there exist some domains in which a particular bias will perform well and some domains where the bias will cause the classifier to perform poorly. The Conservation Law of Generalization Performance explains this by stating, "total generalization performance over all learning situations is null" [Sch94]. In other words, no single learning algorithm can be superior to all other algorithms across all domains. Watanabe's Ugly Duckling Theorem similarly states that it is impossible to perform classification without some sort of bias. [Wat69] Fortunately, researchers in machine learning focus mainly on problems of a relatively limited domain. Because of the Conservation Law of Generalization Performance, when this thesis states that one learning algorithm is superior to another, the statement rather means that the former learning algorithm is superior to another in the particular domain being discussed. Upon further review of Schaffer's conservation law, Rao *et al.* find that the conservation law is only applicable to a uniformly random universe. However, it is highly unlikely that common problems in machine learning are uniformly random therefore we measure performance taking into account the probability that each concept occurs. Using this understanding it is possible that one learning algorithm can be superior to another in our universe [Rao95].

1.1.1 Feature Extraction

Feature extraction is the process of extracting a set of new features from an original set of features through some mapping. [Wys80] Assuming there are n features $A_1, A_2 \dots, A_n$ after feature extraction there exists another set of features $B_1, B_2 \dots, B_m$ where $m < n$ and $B_i =$

$F_i(A_1, A_2 \dots, A_n)$, and F_i is a mapping function. Long searches are typically required to find good transformations, but the goal of feature extraction is to find a minimum set of new features via some transformation function to optimize some performance measure.

The reason for performing feature extraction is that when analyzing complex data one of the major problems encountered is that learning with a large number of variables generally requires a large amount of memory and computation power. Moreover, supervised learning algorithms alone may overfit the training sample and generalize poorly to new samples. Feature extraction is a general term for methods of constructing combinations of the variables to get around these problems while still describing the data with sufficient accuracy.

Several general feature extraction techniques are available. For example, principal components analysis (PCA) is a standard technique in which the original n features are replaced and a set of m new features are generated by linear combinations of the original features. The basic idea is to form an m-dimensional projection (where $1 \leq m \leq n-1$) by the linear projections that maximize the sample variance subject to being uncorrelated with all of the already selected linear combinations. The number of new features m is determined by the m principal components that capture a variance greater than some pre-determined threshold. Finding m principal components can be otherwise thought of as finding the m eigenvectors with the largest eigenvalues that correspond to the dimensions that have the strongest correlation in the data set. [Dan01] Similarly, a feedforward neural network approach can be used to realize a functional mapping and extract new features. The basic idea of this approach is to use the hidden units in the neural network as newly extracted features. The prediction accuracy is used as the performance metric; of course, this requires that the data must be labeled with classes, and, contrary to PCA, the transformation from input units to hidden units is nonlinear. Two types of algorithms have been developed to extract the minimum number of hidden units (and their corresponding features) from neural networks: the network construction algorithm iteratively adds hidden units to improve prediction accuracy, and the networks pruning algorithm removes redundant connections between the input and hidden layers so long as the prediction accuracy does not decrease. [Set01]

1.1.2 Selective Induction

Selective induction, also known as feature selection, is a process that chooses a subset of M features from the original set of N features, where $M \subseteq N$, for the purpose of reducing M according to some criterion [Blu97]. The goal of selective induction in machine learning is to:

1. Reduce the dimensionality of a feature space.
2. Increase the speed of the learning algorithm.
3. Improve the accuracy of the classification algorithm.
4. Improve the comprehensibility of classification results.

Studies have also shown that the reduction in feature spaces provided by feature selection can improve the prediction accuracy in unsupervised clustering algorithms. [Tal99] [Tal99a] [Das00] [DyJ00]

Intuitive, though naïve, ways to generate feature subsets exist. For instance, starting with an empty set we could sequentially add features one at a time and test the efficacy for each subset. If we start with a full set then sequentially we can remove one feature at a time similar to the first scheme. Alternatively, we can randomly generate sets so that each candidate feature has an equal probability of being chosen (from $2^{|N|}$ possible subsets). Finally, we could exhaustively enumerate all $2^{|N|}$ possible subsets. With each scheme a best score is kept, and after the selection process is complete the best feature set is chosen. In all cases, the *best* score is always relevant to a certain evaluation criterion. "Best" can thus be defined as the optimal feature subset for a given criterion.

Evaluation criteria can be categorized into two distinct groups based on their independence from the learning algorithm. A *filter* evaluates the optimality/goodness of a feature subset independent of the learning algorithm by calculating a distance measure, information measure (*i.e.*, entropy), dependency measure, consistency measure, etc. [Liu98]. A *wrapper* evaluates the optimality/goodness of a feature subset by evaluating the performance of the learning algorithm applied on the selected feature subset. For supervised learning, the main goal is to maximize classification/predictive accuracy. While for unsupervised learning many

measures exist for estimating the quality of clustering results, such as cluster compactness, scatter separability, and maximum likelihood. [Tal99a] [DyJ00] [Kim00]

Many feature selection algorithms exist. Using the model described above, we can use one or more types of algorithms to generate a feature subset better suited for classification tasks.

Exhaustive approaches

Starting with an empty or complete feature set we can sequentially add or remove features as described above. Search algorithms, such as branch and bound, evaluate estimated accuracy by starting with a full feature set and running until a preset bound is reached.

Heuristic approaches

Sequential forward search and sequential backward search, described above, can be implemented using many different univariate heuristics. The simplest version of a wrapper model is to run a decision tree learning algorithm and apply only those features that are used, this is effectively using the entropy heuristic because decision trees operate mainly on the entropy score of features. [Car93]

Nondeterministic approaches

The Las Vegas Filter randomly generates and tests feature subsets with an inconsistency measure. [Liu96] Genetic algorithms and simulated annealing are also used in feature selection, but are essentially nondeterministic versions on the heuristic and exhaustive approaches described above.

Instance-based approaches

Relief [Kir92] is a typical example of an instance-based approach. There is no explicit feature subset generation in this approach, rather many small data samples are sampled from the training data and features are ranked based on how well they differentiate instances of different classes for a data sample. Features with a higher score are selected.

As discussed in the above section on biases, if the initial training features are appropriate for the representing the target theories then selective induction works very well in terms of theory complexity and prediction accuracy. However, if the initial training features are not appropriate for describing target theories, then the prediction accuracy and theory complexity performance can quickly deteriorate. This is the fundamental limitation of selection induction algorithms and is one main focuses of this thesis.

In practice, it is very common that domain experts only supply low-level features. An example of a low-level feature for a checkers game domain is the contents of a square on the board. [Sam59] These low level features might contain some information on the state of the game, but there is no direct information simply in the contents of the squares that represents whether or not the game has been won. Even worse, some low-level features might be irrelevant to the target theory, and irrelevant features can often result in worse performance. [Lan94]

There are several other limitations of selective induction, see [Ren90a] [Ren88] [Car94], but the fundamental limitation is that the training features are not always appropriate for learning the target theory. One method for overcoming these limitations is *constructive induction*. Constructive induction algorithms construct new features from among the initial training features and then build theories based on those newly constructed features, sometimes together with the initial training features. In this context, initial training features are known as *primitive features*. The newly constructed features are expected to be more appropriate for learning the target theories than the primitive features from which the new features are constructed.

1.1.3 Constructive Induction

Constructive induction is the process of improving the attribute vector of a learning algorithm in order to make the problem more easily learned for a particular learning algorithm. [Mat89] Constructive induction is often used when the explicitly selected features do not effectively represent the stated problem. For instance, given a machine learning algorithm (L), constructive induction would be appropriate if the training set contains all of the relevant information for the induction of the target function but this information cannot be extracted by L. [Ren90] Moreover, constructive induction is used to deal with learning algorithms, such as feed forward neural networks trained with back propagation and classification and regression tree

(CART) algorithms. [Ben96] [Bre84] Constructive induction generally consists of two parts: one for the construction of new features, and the other for generating theories. After being constructed, new features are treated the same way as the initial, primitive features that were used to construct the new features.

The construction of new features is essentially the application of a set of constructive operators to the set of existing features; this results in the combination, and therefore the construction, of one or more new features. [Mat89] For common learning problems the number of possible constructive operators, such as mathematical operators, set operators, logical operators, etc. and the number of possible constructive operands for each operator is very large, so it is not feasible to search through all possible combinations. This thesis will present an algorithm that does not need to search through all possible combinations.

Common operators include *conjunction* (∧), *disjunction* (∨) and *negation* (-). As an example of their use consider the following features: A_1 (binary), A_2 (nominal with values a, b, and c), and A_3 (continuous), new features $\left((A_1 = false) \wedge (A_2 = b) \wedge (A_3 > 5)\right)$ and $\left((A_1 = true) \vee \overline{(A_2 = b)} \vee (A_3 \leq 5)\right)$ can be constructed. Notice that the first new feature is simply the negation of the second new feature; $(A_1 = false)$ is simply the negation of $(A_1 = true)$.

Take, for example, the decision tree depicted in Figure 1.1. This decision tree shows a fundamental limitation of the selective induction procedure involved in the creation of decision trees called the *replication problem.* [Pag90] Because a decision tree divides each feature space into mutually exclusive regions it is possible to derive duplicate subtrees, such as those shown in the grey regions. If a subtree is replicated many times then many more training examples are needed in order to grow the size of the tree. This often leads to either an overly complex final decision tree, inaccurate pruning of the decision tree, or a premature termination of the learning algorithm. In other words, the replication of subtrees degrades the prediction accuracy of a decision tree learning algorithm.

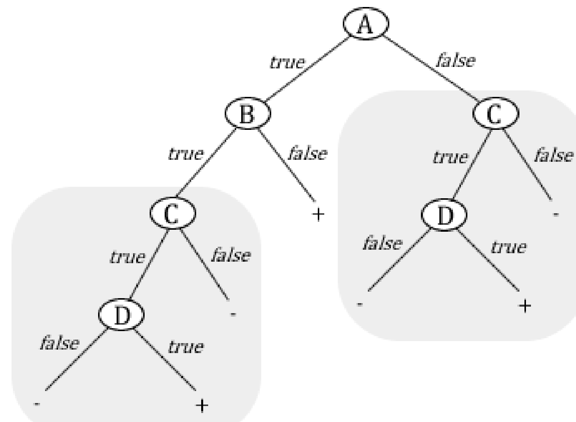

Figure 1.1: An example decision tree for $((A \vee B) \wedge (C \vee D))$

Because the structure of the theory language is fixed, theory learning in the new feature space is expected to be easier than in the original feature space. By constructing new attributes $(A \vee B)$ and $(C \vee D)$, a decision tree can be built for the constructed features $((A \vee B) \wedge (C \vee D))$ whose representation, shown in Figure 1.2: , is dramatically simplified making more complex concepts easier to learn. Therefore, from this point of view, the newly constructed features are more representationally powerful than the primitive features.

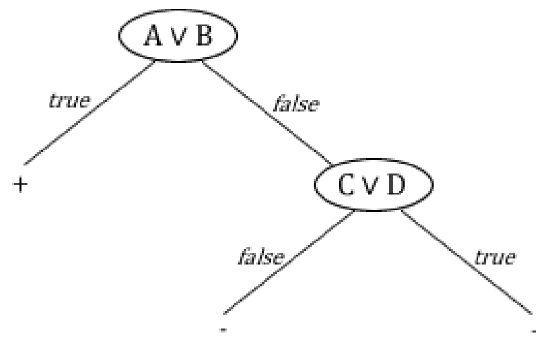

Figure 1.2: Decision tree from Figure 1.1 with two newly constructed features

Motoda and Liu [Mot02] recognize four categories in the constructive induction field.

How to construct new features

Various approaches can be categorized into: data-driven, hypothesis-driven, knowledge-based, and hybrid. The data-driven approach is used to construct new features based on some analysis of the primitive data by applying the available operators. The hypothesis-driven

approach constructs new features based on previously generated hypothesis. The knowledge-based approach constructs new features by applying existing domain knowledge to the set of primitive features. The hybrid approach is a combination of the other three.

How to choose and design operators for feature construction

There are many operators to choose from when combining features. As mentioned above, construction, disjunction and negation are commonly used operators for nominal features. Other operators include M-of-N and X-of-N. [Zhe98] M-of-N is true iff M out of N conditions are true; X-of-N is similar to M-of-N in that X-of-N consists of a set of conditions, but X-of-N states how many conditions in the set are true. X-of-N has ordered discrete values.

How to use operators to construct new features efficiently

The number of possible combinations of all operators and primary features, even in a relatively simple problem, is too large to exhaustively explore every possible combination. It is necessary to find intelligent methods to avoid an exhaustive search. In later chapters, this thesis will show one solution to this problem.

How to measure and select useful new features

Intuitively, not all constructed features are good. In fact, most of the possible combinations of features very poorly represent the target theory. It is thus necessary to be selective when chosing candidate features and operators. One option is to select features by applying the features selection techniques from Subsection 1.1.2 to remove irrelevant or redundant features. If the number of current features is very large it is sensible to make these decisions when a new compound feature is generated to avoid too many features. This requires some indicator as to the validity of the newly constructed feature, some examples measure consistency and distance as used in feature selection.

Constructive induction can be realized in several ways. One commonly used algorithm is greedy search. Greedy search is easily applied to constructive induction tasks on decision trees; the algorithm generates new feature at each decision node based on original features or previously constructed features. To construct a new feature, the algorithm searches greedily

through the instance space using a pre-specified set of constructive operators. Starting from an empty set of decision nodes, the algorithm systematically adds and/or deletes decision nodes until the instance space has been searched or the algorithm is forced to terminate (greedy search is an *anytime algorithm*, meaning it can be stopped at any time and still return valid, although possibly incomplete, results). To evaluate candidate decision trees standard decision tree metrics, such as class entropy and model complexity, can be considered in lieu of test date, or classification accuracy can be used to evaluate the performance of the candidate decision tree in the presence of test data. In the event that all features are numeric, Gama shows that it is useful to search for the best linear discriminant function instead of using standard decision tree metrics. [Gam98]

Finally, constructive induction can be achieved using evolutionary computation approaches such as genetic algorithms or genetic programming. Evolutionary computation algorithms are adaptive search techniques based on the simulation of Darwinian natural selection. This thesis explores the application of the evolutionary computation approach to constructive induction, and will therefore describe the approach in greater depth in Section 1.2.

1.2 Evolutionary Computation

Evolutionary computation is the set of search procedures based on the mechanics of natural selection and natural genetics. That is, they base their approach on Darwin's theory of the survival of the fittest. More specifically, evolutionary algorithms contain some data structure that computationally represents the *genes* of a candidate theory. Operations, based on naturally occurring phenomenon, are applied to the data structure to effectively *evolve* the theory. At the end of each iteration, all of the offspring are evaluated according to a *fitness* function to determine whether the offspring lives or dies. If the offspring dies then it is effectively deleted from the *population*, otherwise, if the offspring lives then it remains in the population where it can reproduce offspring of its own. This basic approach is repeated until the algorithm stabilizes (stop evolving) or until a threshold fitness is reached. Evolutionary algorithms are an anytime algorithm, that is they can be stopped at any time and the best, current result can be returned. Although evolutionary algorithms are randomized, they are not merely random walk. They

efficiently exploit historical information contained within the population to produce a new *generation* of the population with a higher fitness. [Gol89]

The main advantage in evolutionary computation is its robustness. That is, evolutionary algorithms leverage the natural ability of species to repair, regulate and reproduce themselves according to natural pressures. This makes evolutionary algorithms far more robust than traditional search methods.

1.2.1 Traditional Search Methods and Local Maxima

When discussing evolutionary algorithms it is important to note the common alternatives. Calculus-based search methods are the most common alternative. *Hill climbing* is a simple way to find a simple solution. Hill climbing works by starting with an initial structure in the search space (a *point*) and testing the fitness of several alternative structures that are adjacent to the initial point. The adjacent point with the best fitness is always picked and the algorithm repeats itself until it cannot find a better next point. Hill climbing can be easily conceptualized as a stubborn hiker climbing a mountain that refuses to ever decline in elevation. The problem with this approach is that the hiker will never reach the top of the mountain because smaller hills are in the way, and if the hiker can never travel down in elevation then he will become stuck at the top of some small hill. This example accurately describes the problem of *local maxima*. Greedy search algorithms such as hill climbing are destined to be caught in local minima. [Rus03]

Visually, Figure 1.3 shows an example search space gradient for which the hill climbing algorithm would perform very poorly. Starting on the far left of the search space, the hill climbing algorithm would return a theory represented by peak A with a fitness score of 20. A more robust searching algorithm would be able to recognize that peaks A and B are not globally optimal and would instead return the best possible theory represented by peak C.

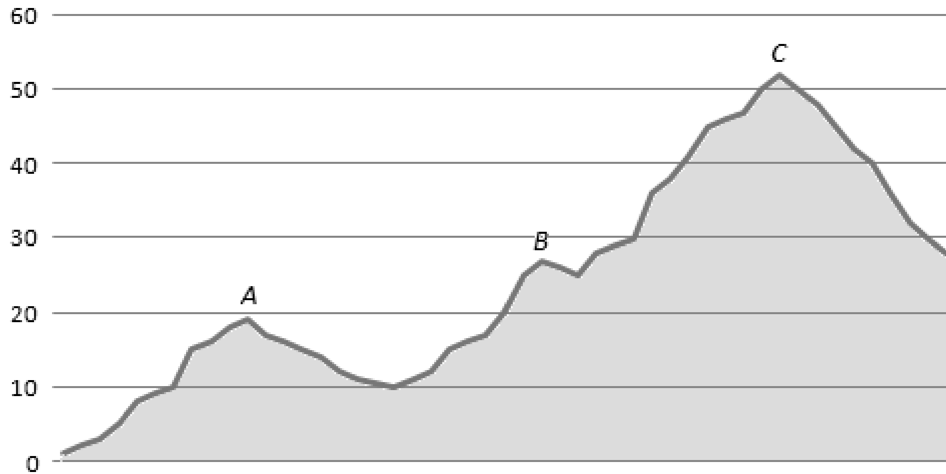

Figure 1.3: Example search space gradient with two local maxima (A, B) and a global optimum (C).

Dynamic programming search algorithms, such as A*-search, are not caught in local maxima. The crux of the dynamic programming approach idea is straightforward: given a discrete, finite search space the search algorithm looks at objective function (or cost function) values at every point in the space, one at a time. The objective function, for instance, of A*-search is the distance traveled plus the estimated distance from the current point to the goal [Rus03]. The dynamic programming approach breaks down, however, if the search space is too large because of its inherent lack of efficiency. Richard Bellman, the inventor of dynamic programming, described this efficiency problem as "the curse of dimensionality" [Bel61], and it is because of this problem that dynamic search algorithms largely cannot be used for the purposes of constructive induction.

Random search algorithms have become popular because they tend not to have the same shortcomings as the calculus-based or dynamic programming approaches. However, random walks through the search spaces that search and save the best still are not efficient enough to process large problems, and in the long run a random search will do no better than the dynamic programming approach. [Gol89] Besides random walks, *simulated annealing* is a hybrid approach that uses random processes to break an otherwise greedy search out of local minima states. [Kir83] Evolutionary algorithms are classified as a randomized search algorithm, but the randomness in this approach references the evolutionary operations that are discussed primarily

in Subsection 1.2.2. It is important to remember that randomized search does not necessarily imply directionless (or blind) search.

Because of the stated deficiencies of traditional search methods, especially in the realm of constructive induction where the search space is very large, evolutionary algorithms are considered more robust.

1.2.2 Genetic Algorithms

The genetic algorithm (GA) is a computational technique that models the evolutionary process in order to solve problems. In GAs, each possible point in the search space of a problem is encoded as a fixed-length bit string (*i.e.*, as a gene). The genetic algorithm attempts to find the best solution to the problem by genetically breeding the population of individuals over a number of generations. According to [Koz98], there are four major preparatory steps required to use the conventional GA:

1. Determine the representation scheme.
2. Determine the fitness measure.
3. Determine the parameters for controlling the algorithm.
4. Determine when to terminate the algorithm.

The representation scheme is essentially what separates GAs from other types of evolutionary algorithms. In a GA, the individuals of a population are usually represented by a fixed-length sequence of bits patterned after biological chromosome strings. In most cases the bits in the sequence are binary making the *alphabet* size equal to 2. Arguably the most important part of the representation scheme is the mapping that expresses each bit-sequence as a point in the search space. This mapping is a fundamental limitation of GAs that is discussed in Subsection 1.2.3.

Even though the representation may be the most important facet of the GA, the fitness function is what drives the evolutionary process. The fitness function evaluates every individual in the population and assigns each individual a score. It is important that a fitness function be able to properly evaluate every possible individual capable of being encountered in the search

space. The fitness function varies widely among GAs, but in most GAs an individual's fitness score is directly proportional to its absolute fitness.

The primary parameters required for controlling the GA are the population size (M) and the maximum number of generations to be run (N). Population sizes can range from a few dozen individuals up to several thousand individuals. The number of possible generations need not be limited; setting a maximum merely stops the GA before the termination criterion is met. Other parameters include probabilities for mutation and crossover, etc.

Termination criteria are necessary for deciding when to terminate and what to do upon termination. An example termination criterion is a fitness threshold or a maximum number of generations (N). The final result (or theory) of the GA is usually the individual in the final generation with the highest fitness.

Once the four preparatory steps are completed the GA is ready to be run. According to [Koz98] there are essentially three execution steps of a GA:

1. Randomly create an initial population of size M with randomly generated bit-sequences.
2. Iteratively perform the following steps until termination criteria is satisfied:
    a. Find the fitness of each individual using the predetermined fitness function.
    b. Create a new population of bit-sequences by applying one or more genetic operations from Subsection 1.2.2.1.
3. Upon termination, find the individual to be designated the final result and return it.

Because the GA is a probabilistic algorithm probabilistic steps are used to determine nearly every step of the algorithm. "Thus, anything can happen and nothing is guaranteed in the genetic algorithm" [Koz98]. Also, due to the nondeterministic nature of GAs, multiple runs are necessary in order to obtain a reliable result.

1.2.2.1 Genetic Operations

There are four biologically recognized genetic operations that contribute to the evolutionary process: *reproduction*, *mutation*, *crossover* and *death*. Genetic algorithms (GAs)

model these biological processes by applying similar transformation to bit-sequences that digitally represent a chromosome. For every individual the sum of the probabilities for all four genetic operations is 1.

Reproduction

Reproduction happens when an individual is probabilistically selected from among the population on the basis of fitness and then copied, without change, into the next generation of the population. The selection is done in such a way that the higher the individuals' fitness the more likely it is to appear in the next generation. As an example of reproduction, consider the bit-sequences from Figure 1.4. The original bit-sequence is simply copied from G_1 to G_2. [Koz98]

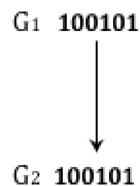

Figure 1.4: Example of reproduction when applied on bit-sequence 100101.

Mutation

Mutation happens when an individual is probabilistically selected from among the population on the basis of fitness. From that individual bit-sequence a *mutation point* is chosen at random and the bit at that location is changed with some mutation probability. The altered individual is then copied into the next generation of the population. Mutation is useful in cultivating diversity among individuals in a population and helps avoid becoming *stuck* in a local optimum as discussed in Subsection 1.2.1. As an example of mutation, consider the bit-sequences from Figure 1.5. The individual in G_1 is probabilistically selected from among the population and then the mutation point is selected at random to be the third bit in the sequence. Next bit, #3 is randomly changed (in this binary alphabet the probability for a bit flip is .5), and the final bit-sequence is copied into G_2. [Koz98]

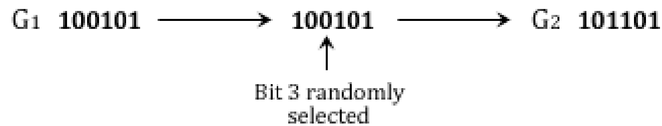

Figure 1.5: Example of mutation occurring on bit #3 in the bit-sequence 100101.

Crossover

The crossover operation allows two new individuals to be created from two parent individuals in order to test new points in the search space. The crossover step starts when two parent-individuals are probabilistically selected from the population on the basis of their fitness.

Individuals can be selected, and usually are selected, multiple times during a generation to produce in reproduction and crossover. When this happens the GA is said to be operating *with replacement*, meaning that after the individual is operated on and children are produced for the next generation the original parent-individual(s) are placed back into the initial population to possibly be selection again. Replacement models the biological notion that more fit individuals are more likely to produce offspring.

After parents (P_1 and P_2) are chosen, crossover begins by randomly selecting two *crossover points*, one from each parent. Each parent is then split into a *crossover fragment* and a *remainder*. The crossover operation then combines the crossover fragment from P_1 with the remainder from P_2 and vice versa to create two offspring. As an example of crossover, consider the bit-sequence from Figure 1.6. The parent-individuals in G_1 are selected and then a crossover point is randomly chosen to occur after the fourth bit. The fragment and remainder pieces are then separated and recombined to form offspring that are copied into G_2.

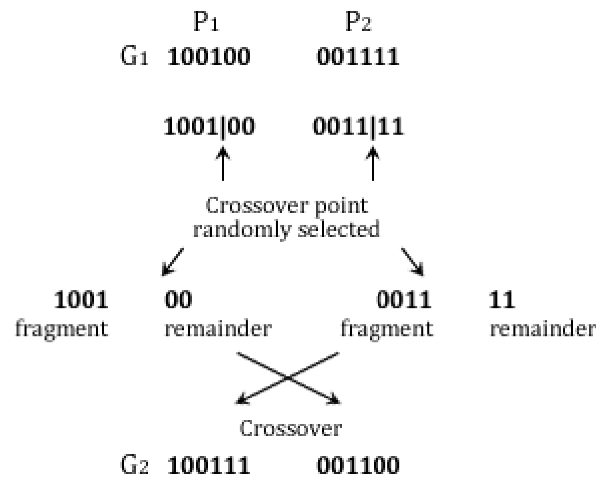

Figure 1.6: Example crossover of the bit-sequences 100100 and 001111.

The two children are usually different from their two parents and from each other. Notice that the child individuals are produced entirely of genetic material from their parents. Therefore, if a bit-sequence representing a relatively fit parent is crossed with another relatively fit parent then it is possible, even likely, that more highly-fit child individuals will emerge as a result of the interaction. When GAs begin to converge on a possible solution, a highly-fit individual begins to dominate the population and crossover operations may involve identical parent-individuals. In this special case, the two child individuals will be identical to each other and identical to their parents. *Premature convergence* occurs when an individual dominates the population but the dominant individual does not represent the globally-best possible individual. [Koz98]

Death

Death occurs when an individual from the initial population is not selected from reproduction, mutation or crossover and therefore does not get included into the next generation's population. That genetic bit-sequence then ceases to exist in the population, although it can be produced again by another genetic operator described above.

1.2.2.2 Probabilistic Selection

Darwin's theory of natural selection is at the heart of evolutionary computation. The ability to select individuals to participate in crossover, mutation and/or reproduction based on their fitness allows the algorithm to keep the good genes and eliminate the bad genes. *Probabilistic selection* is ability that GAs (and other evolutionary algorithms) employ to give

every individual in the population a chance of being selected to participate in one or more genetic operations. Individuals with a higher fitness score duly receive a higher probability of being selected. In this sense GAs are not merely a greedy hill climbing algorithms. Instead, GAs more closely resemble the nondeterministic nature of simulated annealing [Kir83] because individuals with poor fitness still have a certain, although relatively low, probability of being selected.

As an example of probabilistic selection consider a population described in Table 1.1. The population consists of four individuals with their bit-sequences listed in the second column. Each individual's absolute fitness is listed in the third column with a total absolute fitness score of 1079. The corresponding normalized fitness scores are shown in the fourth column as the percentage of the total fitness.

Table 1.1: Example problem individuals with fitness values and corresponding selection probabilities

No.	Individual	Fitness	% of Total
1	011100	150	13.9
2	110011	524	48.6
3	101101	320	29.7
4	100011	85	7.8
Total		1079	100.0

When probabilistically selecting individuals for genetic operations each individual's normalized fitness is conceptually placed on a roulette wheel, as shown in Figure 1.7, the result is a *weighted roulette wheel*, where the probability of selection for an individual is conceptually equal to the probability of the corresponding individual being chosen from the roulette wheel. To reproduce, simply spin the weighted roulette wheel and select the resulting individual. In the case of crossover operations, two individuals are required and so two spins are required. [Gol89]

Selection without replacement occurs if the selected individual is removed from the selectable population and the remaining individuals are reweighted. Selection with replacement occurs when the selected individual is not removed from the selectable population and probabilities remain the same. Intuitively, evolutionary systems that perform selection with

replacement can operate on the same individual multiple times, while evolutionary systems that perform without replacement cannot select the same individual more than once.

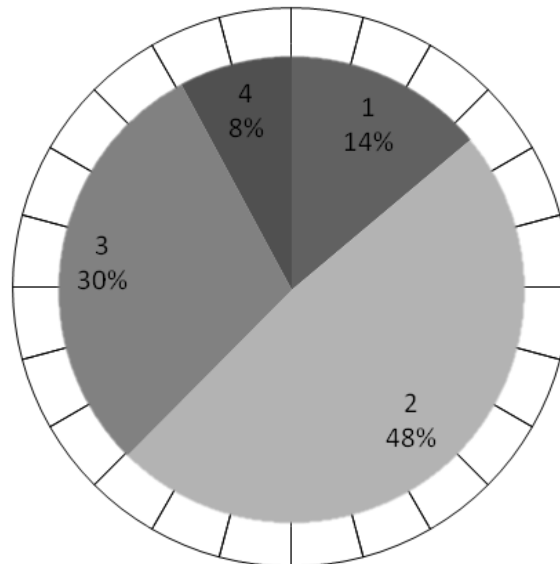

Figure 1.7: Roulette wheel visualization for normalized fitness

From the above content the genetic algorithm according to Perry is shown in Figure 1.8. [Per03]

```
1. generate initial population;
2. do{
3.    evaluate fitness for all individuals;
4.    select best individuals;
5.    perform genetic operations to create next generation;
6.    increment currentGeneration;
7. } until (currentGeneration = maxGenerations);
```

Figure 1.8: Pseudocode for a genetic algorithm.

1.2.3 Genetic Programming

Genetic programming (GP) is an extension of the genetic algorithm (GA) described in Subsection 1.2.2 in which the structures that are operated upon vary in size and shape, and have expression tree semantics. The goal of a GP is the same as that of a GA in that GPs non-deterministically search for a solution to a problem using the principles of Darwinian natural

selection. Therefore this section will focus only on the issues that are distinctly related to GP; for a broader survey on evolutionary computation see Subsection 1.2.2.

Before a GP can begin five preparations must be made. These involve determining:

1. The set of terminals
2. The set of primitive function
3. The fitness measure
4. The parameters for controlling the algorithm
5. When to terminate the algorithm

Notice that steps 3, 4 and 5 are identical to steps 2, 3 and 4 from the genetic algorithm set up from Subsection 1.2.2. Therefore this subsection will only address steps 1 and 2.

The first major step in preparing a GP is to identify the *terminal set* for the problem. Terminals correspond to the features from the search space.

The second major step in preparing a GP is to identify the *function set* for the problem. The functions are typically arithmetic operators such as $+$, $-$, etc., logical operators such as $\wedge$, $\vee$, etc., set operators such as $\cap$, $\cup$, etc., domain specific operators, or any combination of one or more types of operators. These functions should be able to accept any terminal presented to them during computation, and, furthermore, each function should be able to recursively accept every possible function containing any number of possible sub-functions containing any defined terminals, etc. A function set and a terminal set that together satisfy this requirement are said to have *closure*. [Koz98]

1.2.3.1 Representation

As previously stated, the main difference between GAs and GPs is the representation of the feature space. Because of the combination of functions and terminals involved GPs are typically represented as trees where functions are always found at branch nodes and terminals are always found in the leaves.

As an example, consider the following random equation: $(Z-4)X+Y(W+9)$. As an individual in a GP this example equation would be represented as a tree where the functional operators $\{*, -, +\}$ are branch (*i.e.*, non-leaf) nodes in the tree and the variables and literals $\{\mathrm{Z}, 4, \mathrm{X}, \mathrm{Y}, \mathrm{W}, 9\}$ are leaf nodes, and therefore terminals, as shown in Figure 1.9.

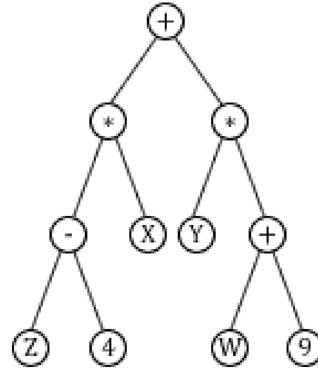

Figure 1.9: Tree representation of the equation $(Z-4)X+Y(W+9)$ for an individual in a genetic program.

1.2.3.2 Genetic Operations

As previously illustrated in Subsection 1.2.2.1, there are four primary genetic operations that can be performed. They are reproduction, mutation, crossover and death. Reproduction occurs when one individual from the current generation is copied to the next, and death occurs when an individual is not operated on at all. These simple genetic operations behave the same in GAs as they do in GPs so no further explanation is required. However, mutation and crossover operations behave differently because of the different representation semantics.

Mutation

When an individual is probabilistically selected for mutation a node (branch or leaf) is selected at random and is replaced by another random subtree. As an example, suppose the individual from Figure 1.9 is selected for mutation. As shown in Figure 1.10, the node representing the $+$ operator is randomly chosen from tree A. Tree B shows that the selected node and all children-nodes are deleted. Tree C is randomly generated and appended to the original

tree to form tree D. The final tree represents the newly formed equation: $\left((\mathrm{Z}-4)\mathrm{X}\right)+\left(\mathrm{Y}\left(\mathrm{Y}(4-\mathrm{X})\right)\right)$. [Koz98]

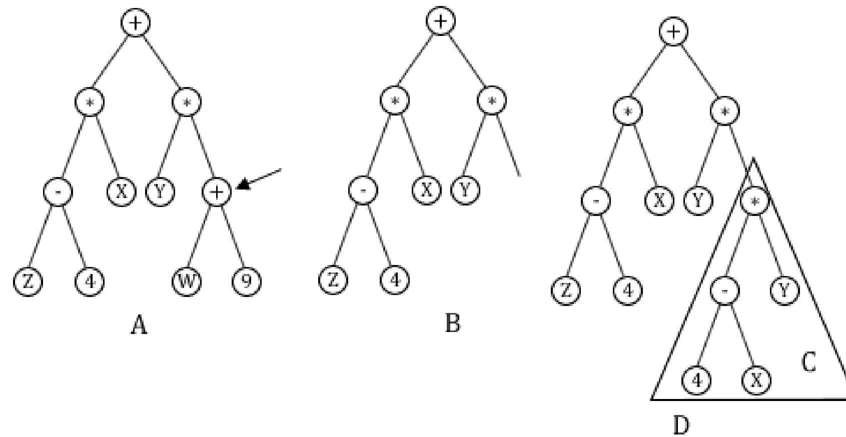

Figure 1.10: Example of GP mutation where the individual from Figure 1.9 is mutated to form a new individual.

<u>Crossover</u>

As in GAs, crossover in GPs requires two parent individuals to donate a part of themselves to be crossed with the other. Also as in GAs, two parents are probabilistically selected. Then each parent randomly selects a crossover node. The resulting crossover fragments are sub-trees where the selected crossover node is the root of the fragment. The remainders are the original individuals with the selected fragment removed. The fragments are crossed and rejoined with the remainders in the empty spot to form two offspring that are copied into the next generation. For example, Figure 1.11 shows two probabilistically selected parent-individuals (Parent 1 and Parent 2) which have each had crossover points identified. Then the individuals are split into remainder and fragment pieces. Finally, the fragments are crossed and reattached to the opposing remainder piece to form two new offspring. Specifically, this example crosses the parent-equation $(\mathrm{Z}-4)\mathrm{X}+\mathrm{Y}(\mathrm{W}+9)$ with the parent-equation $(4-\mathrm{X})\mathrm{Y}$ to form offspring equations $(\mathrm{Z}-4)\mathrm{X}+\mathrm{Y}(4-\mathrm{X})$ and $(\mathrm{W}+9)\mathrm{Y}$. This effectively crosses the term $\mathrm{W}+9$ from the first parent with the term $4-X$ from the second parent.

As in GAs, it is possible for a GP to select the same individual as both parents. One difference between GPs and GAs is that the crossover points are selected at random

independently of one another. In other words, the crossover point for Parent 1 is not necessarily the same as the crossover point for Parent 2. Therefore, in GPs it is not necessarily the case that crossover between twin individuals will result in twin children. Because the crossover operation results in children that contain only genes (*i.e.*, data) from their parents the crossover operation always produces syntactically valid individuals [Koz98].

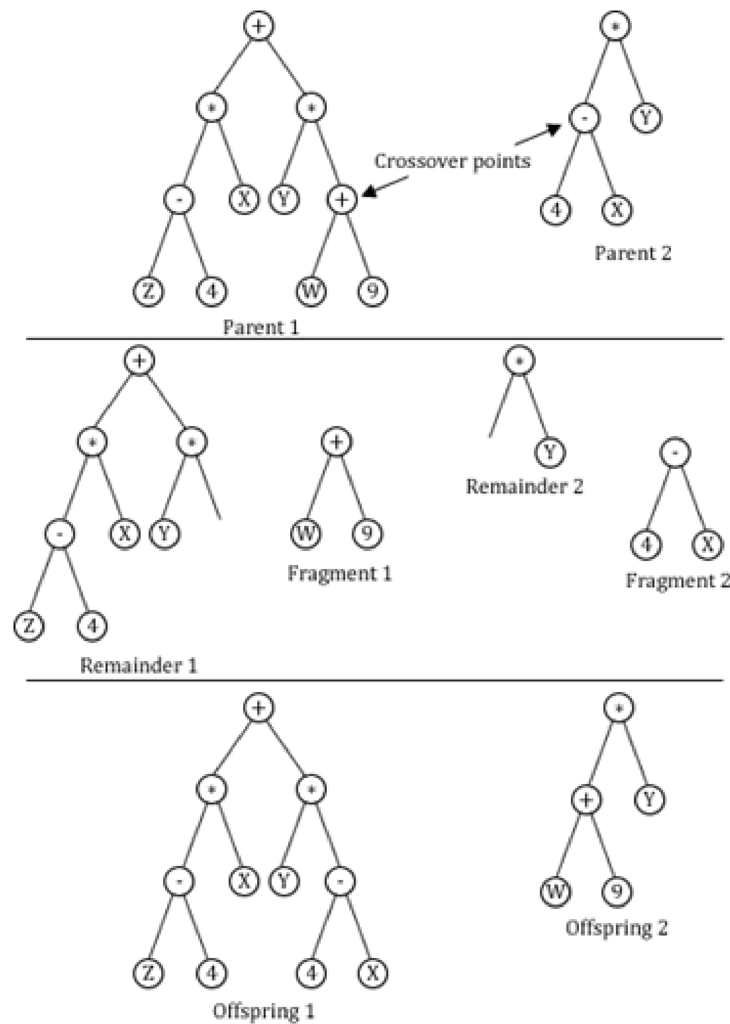

Figure 1.11: Example of GP crossover where two parent individuals are split and recombined to form two new individuals.

1.3 Graph Theory

One goal of this thesis is to study the relationships between entities. For example, in a network of friends people are entities and the relationships between people-entities are often called friendship. Situations such as this are easily described graphically, with points (called *vertices*) representing the entities and lines (called *edges*) representing the relationships. [Col92]

A *graph* consists of a collection of vertices and a collection of vertex pairs called edges. For example, consider a graph ($\mathcal{G}$) with a set of five vertices ($\mathcal{N} = \{n_1, n_2, n_3, n_4, n_5\}$) and a set of five edges $\left(\mathcal{L} = \{\{n_1, n_2\}, \{n_1, n_3\}, \{n_2, n_4\}, \{n_3, n_4\}, \{n_3, n_5\}\}\right)$. $\mathcal{G}$ is expressed graphically in Figure 1.12 [Was07].

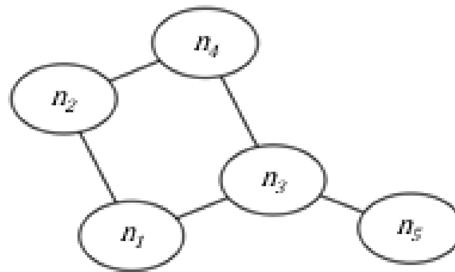

Figure 1.12: Example undirected graph with five vertices and five edges.

Two vertices are said to be *adjacent* if they form an edge. For example, in Figure 1.12 n_1 and n_3 are adjacent. Adjacent vertices are also called *neighbors*. A *path* is a list of vertices in which each successive pair is an edge. For example, in Figure 1.12, n_1, n_3, n_5 is a path from n_1 to n_5. Another path from n_1 to n_5 is n_1, n_2, n_4, n_3, n_5 [Col92]. The distance from one vertex to another is the number of vertices that are in the path from n_i to n_j. For example, the shortest distance from n_1 to n_5 is 3 and the corresponding shortest path is n_1, n_3, n_5. The *degree* of a vertex, denoted by $d(n_i)$, is the number of vertices that are adjacent to it. For example, in Figure 1.12 $d(n_1) = 2$ and $d(n_5) = 1$.

One important note about the type graphs that have been described in this section so far is that the edges of the graph are non-directional; specifically, this type of graph is said to be an undirected graph.

1.3.1 Directed Graphs

Many relations are *directional*, which means that the ties are oriented from one vertex to another and the relation is not necessarily reciprocated. The import and export of goods from one nation to another is an example of a directional relation while the more simple relationship of trade is undirected. This thesis deals only with directional relations that are presented in directed graphs, or *digraphs*.

As in the case of the graph discussed in Section 1.3, the digraph $\mathcal{G}_d(\mathcal{N},\mathcal{L})$ consists of two sets: a set of nodes $\mathcal{N} = \{n_1, n_2, \ldots, n_g\}$ with a size of g, and a set of directed edges or *arcs* $\mathcal{L} = \{l_1, l_2, \ldots, l_L\}$ with a size of L. Each arc is an ordered pair of vertices such that $l_k = \langle n_i, n_j \rangle$. The arc $\langle n_i, n_j \rangle$ is directed from n_i (the origin) to n_j (the sink or receiver).

In a digraph, a vertex can be either *adjacent to* or *adjacent from* another node depending on the direction of the arc. Therefore the *indegree* of a vertex, $d_I(n_i)$ is the number of vertices that are adjacent *to* n_i. The indegree of node n_i is equal to the number of arcs where $l_k = \langle n_j, n_i \rangle$ for all $l_k \in \mathcal{L}$ and all $n_j \in \mathcal{N}$. The *outdegree* of a vertex, $d_O(n_i)$ is the number of vertices that are adjacent *from* n_i. The outdegree of node n_i is equal to the number of arcs where $l_k = \langle n_i, n_j \rangle$ for all $l_k \in \mathcal{L}$ and all $n_j \in \mathcal{N}$ [Was07].

As an example, consider the digraph shown in Figure 1.13. Mathematically, the digraph is represented by $\mathcal{G}_d(\mathcal{N},\mathcal{L})$ with $\mathcal{N} = \{n_1, n_2, n_3, n_4, n_5\}$ and $\mathcal{L} = \{\langle n_1, n_2 \rangle, \langle n_2, n_4 \rangle, \langle n_4, n_3 \rangle, \langle n_3, n_4 \rangle, \langle n_5, n_3 \rangle\}$. In the same example, $d_I(n_1) = 0$, $d_I(n_3) = 3$, $d_O(n_1) = 2$, and $d_O(n_3) = 1$.

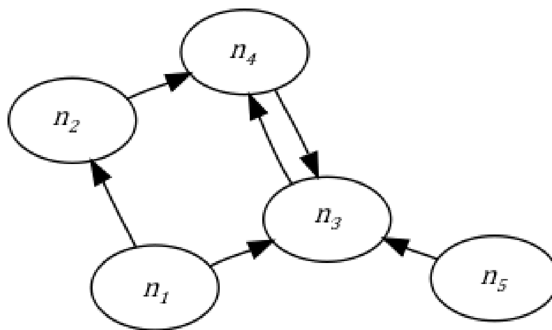

Figure 1.13: Example digraph with five vertices and six arcs.

Moreover, in this directed graph there is no path from n_1 to n_5 because n_5 has no incoming arcs. Likewise there is no path from n_5 to n_1.

1.3.2 Social Networks

Interest in social networks has grown rapidly in recent years. This growth can be attributed to an increasing sophistication in the technical tools available to users. Web sites such as, *Facebook*, *MySpace*, *LiveJournal* and others have each in their own way contributed to the popularity of social networks. The use of these social networking sites has drawn attention from researchers who look to study social networks because these sites provide a means for explicitly stating the relationships between people. Before the advent of social networking sites, researchers relied upon surveys of relatively small groups of people to develop their theories.

As collections of explicit social network information began to grow, so did the scale of social network research. Epidemiologists realized that epidemics do not progress uniformly through populations [Mor93], biologists began to use methodologies from social networks to analyze protein interactions [Par07], counterterrorist organizations began model terrorist networks [Kre02], and so on [Was05].

The analysis of social networks is one perspective methodology that is based on the idea that society can be modeled as a group of relationships between people. Therefore, every social phenomenon can be described in relational and social terms with the condition that the structure of the phenomenon can be expressed in terms of persons and relationships of varying nature between those persons. [Dam08]

Social network services such as *MySpace* and *Facebook* allow users to list interests and link to friends, sometimes annotating these links by designating trust levels or qualitative ratings for selected friends. Some such services, such as Google's *Orkut*, are community-centric; others, such as the video blogging service *YouTube* and the photo service *Flickr*, emphasize social media; while some, such as Six Apart's *LiveJournal* and *Vox*, are organized around text-and-image weblogs. *LiveJournal* and its derivative services, such as *InsaneJournal*, *DeadJournal*, and *JournalFen*, are based on the same open-source server code. In 2008, there are over 17.5 million *LiveJournal* accounts, 1.9 million of them active. This thesis studies the friends network

of *LiveJournal*, which has two varieties of accounts: users and communities (omitting RSS feeds). One advantageous property of its data model, stemming from a common schema for the two account types (which could originally be converted from user to community), is that it provides a simple, flexible representation for entities and relations.

Table 1.2: Types of link in the blog service *LiveJournal*

Start	End	Link Denotes
User	User	Trust or friendship
User	Community	Readership or subscribership
Community	User	Membership, posting access, maintainer
Community	Community	Obsolete

Table 1.2 shows the types of links in *LiveJournal* and their constituent attributes. Friendship is an asymmetric relation between two accounts, each represented by a vertex in a directed graph. The type of the start and endpoint defines the relationship set attributes of the link. For example, a user u who adds another user v to his or her friends list can specify the membership in any of up to 30 groups. These serve the dual purpose of blog aggregation (posts from each group's members are filtered into its aggregator page, which u can read or make public) and groups-based security (each group denotes a read/comment access control list). Access control lists for communities are associated with memberships (community-to-user links), while content is controlled by posters or subscribers. Acquisition of privileges is a community property, of which only membership may be acquired solely by user action ("joining" a community), if the moderator has specified open membership.

Thus, a reciprocal link between a user and a community means that the user both subscribes to the community and is an approved member. Links from user u to v are listed in the "Friends" list of u and in an optionally displayed "Friends Of" list of v. This list can be partitioned into reciprocal and non-reciprocal sublists for a user u:

Mutual Friends: $\{v | \langle v, u \rangle \in \mathcal{L} \land \langle u, v \rangle \in \mathcal{L}\}$
Also Friend Of: $\{v | \langle v, u \rangle \in \mathcal{L} \land \langle u, v \rangle \notin \mathcal{L}\}$

The community analogue of the "Friends Of" list is the "Watched By" (subscriber) list, whose members have the community name listed in the "Friends: Communities" sections of their

individual user profile pages. The community analogue of the "Friends" list is the "Members" list.

The friends network for *LiveJournal* consists of a very large central connected component and many small islands, most of which are singleton users. There are a few source vertices, corresponding to accounts that link to others but have no reciprocated friendships; these are usually RSS or blog aggregator accounts owned by individuals. Additionally, there are sink vertices corresponding to accounts watched by others, but which have named no friends. Some of these are channels for announcement or dissemination of creative work. [Hsu07]

1.3.3 Link Mining

Analysis of friends networks provides a basis for understanding the web of influence [Kol01] in social media. In particular, the problems of determining the existence of links and of classifying and annotating known links are first steps toward identifying potential relationships. This inferred information can in turn be used to introduce new potential friends to one another, make basic recommendations such as community recruits or moderator candidates, or identify whole cliques and communities.

In 2006, Hsu *et al*. introduced a link prediction problem for *LiveJournal*: given a graph in which the existence of a candidate link is hidden (elided if it exists), classify it as present or absent given all other attributes of the graph and of the endpoints. Hsu's initial approach to link identification consisted of dividing friends network features into graph features and interest-based features. [Hsu06]

Graph features could be computed simply by scanning the graph, in the case of pair-distance metrics, performing all-pairs shortest path (APSP) search:

1. Indegree of u, *i.e.*, $d_I(u)$: popularity of the user
2. Indegree of v, *i.e.*, $d_I(v)$: popularity of the candidate
3. Outdegree of u, *i.e.*, $d_O(u)$: number of other friends besides the candidate; saturation of friends list

4. Outdegree of v, *i.e.*, $d_O(v)$: number of existing friends of the candidate besides the user; correlates loosely with likelihood of a reciprocal link
5. Number of mutual friends w s.t. $\langle u, w \rangle \in \mathcal{L} \wedge \langle w, v \rangle \in \mathcal{L}$
6. "Forward deleted distance": minimum alternative distance from u to v in $\mathcal{G}_d$ without the edge $\langle u, v \rangle$
7. Backward distance from v to u in $\mathcal{G}_d$

These were supplemented by interest-based features:

8. Number of mutual interests between u and v, *i.e.*, $|\mathcal{I}_u \cap \mathcal{I}_v|$
9. Number of interests listed by u, *i.e.*, $|\mathcal{I}_u|$
10. Number of interests listed by v, *i.e.*, $|\mathcal{I}_v|$
11. Ratio of the number of mutual interests to the number listed by u, *i.e.*, $\frac{|\mathcal{I}_u \cap \mathcal{I}_v|}{|\mathcal{I}_u|}$
12. Ratio of the number of mutual interests to the number listed by v, *i.e.*, $\frac{|\mathcal{I}_u \cap \mathcal{I}_v|}{|\mathcal{I}_v|}$

1.4 Principal Claims

In this thesis, I consider the problem of discovering links in a large, incomplete graph. This thesis presents an approach to link prediction that is based on the combinations of graph feature analysis and intrinsic attributes of entities. Therefore the principal claims of this thesis are:

1. By crawling the social network service *LiveJournal* an appropriately large graph can be realized and learnable features can be ascertained.
2. Feature analysis can be achieved on a very large graph by the efficient management of the underlying data structure.
3. Operators within the genetic programming approach can be used to construct new features from primitive features that provide a more learnable description of the graph.
4. Features constructed from a genetic program will improve the performance of learning algorithms for link mining.

Therefore, this thesis asserts that link discovery in very large graphs by constructive induction using genetic programming achieves better results than previous attempts that do not consider the entire graph and/or do not employ genetic programming.

CHAPTER 2 - Related Research

In the previous chapter I introduced the four broad concepts that encompass this thesis, as well as stated the principal claims of this work. This chapter will discuss current and continuing research in link discovery and constructive induction in order to give a context for my original work which is described in the following chapters.

2.1 Meta Learning

Meta learning studies how learning algorithms and other systems can increase in efficiency with experience. The goal of meta-learning is to understand how learning occurs and then use that information to improve the learner. [Vil02] The traditional learning algorithm, discussed in Section 1.1, differs from meta-learning in that a meta-learner discovers the learning bias dynamically by searching for the best learning strategy as the algorithm progresses. [Thr98]

The proper meta-learning system would begin with a certain *base learner*, *i.e.*, a traditional learning algorithm with a fixed bias. Once the system is started and meta-information about the state of the learner begins to accumulate, the meta-learner is able to change its bias by

switching to another base learner [Wol92]. A more "granular" [Vil02] approach consists of selecting a learning algorithm for each individual training example. That is, if meta-data is gathered that can be applied to discriminate different classes of examples then the best base-learner can be chosen and applied to each particular example [Mer95]. The algorithm selection is done based on its performance on each class of examples.

Meta-learning is similar to constructive induction using genetic programming because they both leverage information about the current state/progress of the system in order to make adjustments to the learning process. Meta-learning does this in the manner described above, while genetic programming does this by constructing new features based on previous features recursively.

Despite the current research efforts and promising results thereof, meta-learning is not a candidate for this thesis because it will not necessarily result in a generalized theory. This is because the different base learners will all individually construct their own theories based on their inherent biases. While meta-learning is useful in leveraging those biases, I seek a single, generalized theory about my examples.

The following subsections describe three popular meta-learning approaches that are used for comparison in this thesis.

2.1.1 Bagging

Bagging [Bre96] is an iterative approach that creates a set of classifiers such that each classifier is trained by a random redistribution of the original training set (with replacement) and each random redistribution has the same number of examples. For example, a set of training examples $X = \{x_1, x_2, x_3, x_4\}$ can have redistributions such as $D_i = \{x_1, x_1, x_2, x_3\}$ or $D_j = \{x_4, x_4, x_4, x_4\}$, but $|D|$ is always equal to $|X|$. On average, D_i will have 63.2% of the original training data in it because each feature has a $1 - \left(1 - \frac{1}{N}\right)^N$ probability of being selected for inclusion in D_i. Iteratively, all test examples are evaluated on all classifiers and in each instance the classifier with the highest performance on that specific test example is used.

Bagging, essentially, improves generalization error by reducing the variance of the base learner. That is, if the base learner is unstable then bagging can help reduce the errors associated with random fluctuations in the training data. However, if the base learner is already stable then bagging can actually degrade performance because the base learner is using only 63.2% of the total training data. [Tan06]

2.1.2 Boosting

Boosting [Fre96] is a specific, formally proven approach to meta-learning that is based on the observation that finding many low quality hypotheses can be a lot easier than finding a single, highly accurate hypothesis. The boosting algorithm works by training a base learner repeatedly, each time feeding it a different subset of the training examples. Each time the base learner is called a new, weak hypothesis is produced. After many iterations, the boosting algorithm combines the weak hypothesis into a single hypothesis that is hopefully more accurate than any of the weak hypotheses.

To make this approach work two questions must be answered: (1) how should the training examples be chosen each round, and (2) how should the weak hypotheses be combined into a single strong hypothesis? Regarding question one, generally boosting algorithms choose examples that are most often misclassified by preceding attempts. In effect, this forces the base learner to focus most on the difficult examples. Regarding question two, the final or combined classifier is a weighted majority vote of the base classifiers where the weights are an inverse function of the base classifiers' error. [Sch02]

2.1.3 Random Forests

Finally, Random Forests [Bre01] is a meta-learning approach which constructs many classifier trees (*i.e.*, a forest) and combines them into a single decision tree. The algorithm is as follows. Let the number of training cases be N, and the number of features be M. The number of decisions m at each node of the tree is explicitly given; $m \ll M$. Next a training set is chosen by choosing N times with replacement from all N training cases, exactly as in bagging (see Subsection 2.1.1). For each node in the tree, m features are randomly chosen and the best split on these features is calculated. This split calculation is usually based on entropy.

According to a website on random forests by its creator, Leo Breiman's [Bre02], "[random forests] is unexcelled in accuracy among current algorithms." Empirical studies [Tan06], [Cha08] show that random forests typically provide better performance accuracy and the algorithm is much faster than bagging alone or boosting.

2.2 Krawiec's Feature Construction with Genetic Programs Approach

In Krawiec's work [Kra02], a genetic program (GP) is used to change the representation of the input data (*i.e.*, training and test examples) for machine learning algorithms. Specifically, the author first proposes the general framework for GP-based feature construction. The author also proposes an extended approach that preserves useful features from being evolved as opposed to the standard approach where valuable features can be lost during search. Figure 2.1 shows Krawiec's feature construction model graphically.

Krawiec uses the ECJ software package [Luk01], and the function set included $+$, $-$, $\times$, %, log $<, >, =$, and an approximate equality operator. The terminal set contained the original features. Individuals' fitness was evaluated by running 5 independent 2-fold cross validation runs on the training set. The WEKA [Wit99] implementation of the decision tree inducer C4.5 [Qui92] is the learner used from training and testing.

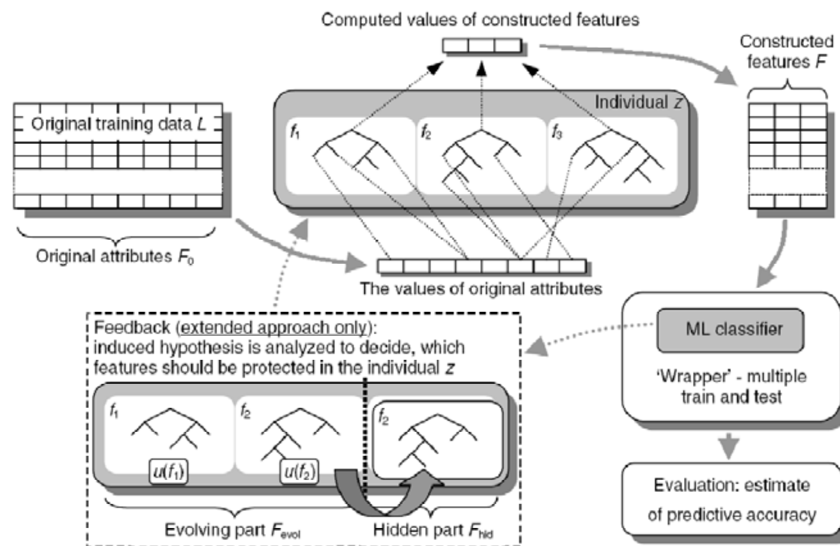

Figure 2.1: Taken with permission from the feature construction Genetic programming-based framework for **[Kra02]**.

Results of Krawiec's experiments show "remarkable" gains in predictive accuracy. The GP-constructed features always resulted in an accuracy which was better than or equal to the original, unconstructed features. Despite the success of his methods, the extended approach did not outperform his unextended approach. Krawiec concludes that GP-based construction of features is a promising endeavor, but more research is necessary "to get rid of some weak points of the method" [Kra02].

2.3 Hsu's Feature Selection for Link Mining Approach

Hsu *et al*. propose an approach for link mining that effectively recommends relationships (*e.g.*, friendship) where none currently exists, or predicts the existence of relationships if and when they are hidden. [Hsu06], [Hsu07] The approach uses graph features, such as indegree and outdegree, pair-dependent features, such as the number of mutual friends and the distance between users, and user-only features, such as number of interests, number of friends, etc. A 941-node graph was computed from a short crawl of *LiveJournal* and the appropriate features were computed. To train a classifier, links were explicitly cut between candidate pairs so that a distance of 1 does not automatically reveal the friendship and the features were fed to three learning algorithms.

Hsu *et al*.'s approach obtained a very high test-set accuracy of 98.2% using the J48 classifier on their graph. More interestingly, when distance features were removed the prediction accuracy dropped severely to 94.8%. Furthermore, when only interest features were used the prediction accuracy dropped even further to 88.5%. This shows that the inclusion of graph features and pair-dependent features significantly improves the prediction accuracy. The authors conclude that their feature selection methods coupled with a good inducer can effectively predict relationships.

CHAPTER 3 - Methodology

This chapter presents my approach for link discovery in very large graphs by constructive induction using genetic programming. This is done in several parts. First, I present a web crawler that gathers semi-structured user information from *LiveJournal* and constructs a large directed graph representing that data. Second, I select base features from the gathered data. Third, I generated 3 sets of candidate pairs. Fourth, I developed genetic program which takes as input the aforementioned base-features and constructs new features. Fifth, I developed algorithms that compute those newly-constructed features for each candidate pair. Finally, I discuss the experimental design for this research with descriptions of the standard evaluation metrics.

3.1 Crawling a social network

Before learning can begin an appropriately sized and realistically distributed example set needs to be obtained. For this thesis, I choose to *crawl* the social network *LiveJournal* for user information and relationships. Toward that end, the fourth generation of the *LJCrawler* application was developed and executed. LJCrawler has essentially five modules: (1) Multi-threaded downloader, (2) text extraction, (3) Storage, (4) URL Queue, and (5) Scheduler. This

section will discuss my implementation of these parts before presenting some of the crawl results.

Proper web crawlers need to be able to handle multiple requests at once. This is because the bandwidth usage of a single web-document download does not fully utilize most systems' capabilities. Any software system that is capable of spawning sub-processes to handle simultaneous tasks is said to be multi-threaded. In the web crawling domain individual threads simultaneously request web pages so a higher throughput can be achieved. LJCrawler uses the multi-threaded approach by keeping a *thread pool* which maintains the threads. Due to the terms of *LiveJournal's* bot policy[1], a crawler may not request more than 5 pages per second (pps). Therefore, the thread pool is initially given a maximum of 7 threads. During execution the thread controller monitors the download rate and dynamically adjusts the maximum number of threads. For example, if the thread controller finds that the crawler is retrieving pages at 3pps then the controller will increment the maximum size of the thread pool by 1. Inversely, if the thread controller finds that the crawler is retrieving pages at 7pps then the controller will decrement the maximum size of the thread pool by 1 effectively throttling the system. The dynamic multi-threading property of LJCrawler keeps the system crawling as fast as *LiveJournal's* terms allow regardless of the wide fluctuations often seen in Internet bandwidth and latency. The dynamic thread controller algorithm is shown in Figure 3.1.

```
1.  int MAX_RATE = 5;
2.  int maxThreads = 7;
3.  ThreadPool pool = new ThreadPool(maxThreads);
4.  while (stillCrawling)
5.     float rate = determineCrawlRate();
6.     if (rate.isBetween(MAX_RATE-1, MAX_RATE) )
7.        //do nothing
8.     else if (rate > MAX_RATE)  //too fast
9.        maxThreads--;
10.    else  //too slow
11.       maxThreads++;
12.    pool.setMaxThreads(maxThreads);
```

Figure 3.1: Pseudocode for dynamic thread control in LJCrawler

[1] *LiveJournal's* bot policy is available at `http://www.livejournal.com/bots/`

User information is kept in two forms on *LiveJournal*: in HTML profile pages and in friend of a friend (FOAF) files. Profile pages are available from `http://username.livejournal.com/profile?mode=full`. FOAF pages are available as XML documents from `http://username.livejournal.com/data/foaf`. Earlier versions of LJCrawler attempted to extract content from the profile pages, but that proved to be difficult and time consuming because it is difficult to automatically extract structured content from within messy HTML pages. The current version of LJCrawler downloads the structured FOAF pages and uses a visitor pattern to extract the content from the XML. This is much easier to do because the FOAF schema[2] is readily available and rarely changes.

Once the data is extracted from the FOAF document it is sent to storage. In order to store the vast amounts of data being gathered by LJCrawler, a data model is needed to be able to express to graph-nature of *LiveJournal* while accurately storing and retrieving the previously extracted data quickly. After several attempts at alternative storage approaches, *e.g.*, Lucene [Gos05], and the Berkeley Database [Ols99], the relational database MySQL was chosen to store user-data and the corresponding structure.

Because a social network is essentially a graph of users, the database schema, shown in Figure 3.2 has a VERTICES table and an ARCS table. The VERTICES table stores the user-independent information while the ARCS table essentially stores a pair of users representing a directed edge in the graph. The PERSON_INTERESTS and INTERESTS tables are a normalization of the interest content from the VERTICES table.

A seed user u_s is given when LJCrawler is first started. That user's FOAF page is the first to be accessed and downloaded. u_s is added to VERTICES and VERTICES.CRAWLED is set to TRUE because u_s is being crawled. Once u_s is downloaded, user information is extracted including friends information. The friends V_s of u_s are then added to VERTICES and $\forall_{v_i \in V_s}(\mathrm{VERTICES.CRAWLED} = \mathrm{FALSE})$ to indicate that they have not yet been crawled. Threads are now able to query the VERTICES table to get uncrawled users. As soon as a thread retrieves the first of the uncrawled users, it takes control of it by setting VERTICES.CRAWLED to TRUE so that another thread does not crawl it. Also note that VERTICES.NICK is an index and is therefore

[2] FOAF schema is available at `http://xmlns.com/foaf/0.1/`

unique; this prevents the same user to be inserted into `VERTICES`, and therefore crawled, more than once. In this manner the `VERTICES` table operates as a queue with a cursor at the end of the table inserting unique users, and a cursor iteratively providing the thread pool with new users at their convenience.

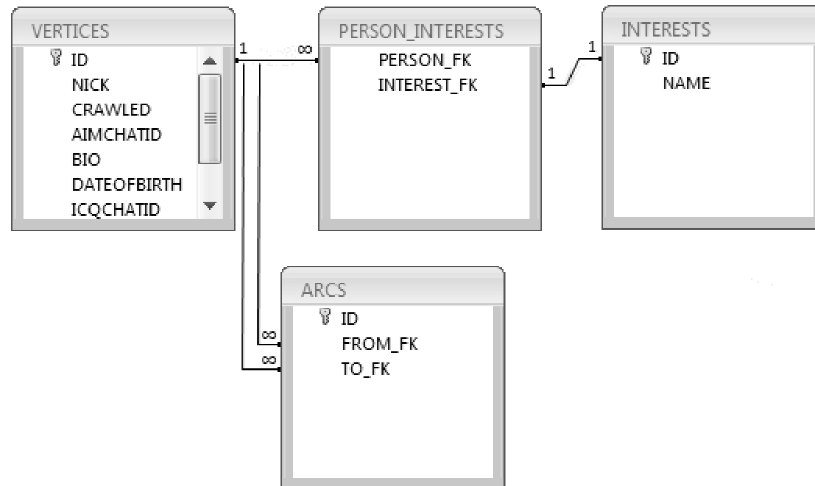

Figure 3.2: Storage schema for MySQL database

The database queue does not have a response time as fast as a similar data structure operating in memory; however, the sheer amount of data would quickly overrun the limits of even a large server's memory capacity. Despite the obvious speed drawback, the crawler's main bottleneck is still the limit of 5 pages per second imposed by *LiveJournal*. In other words, the database usage provides a much needed storage solution without a noticeable decrease in performance.

3.1.1 Crawl Results

LJCrawler was executed once at 6:41pm CST on August 13, 2008 and was let run until 9:10pm CST on Kansas State's Beocat cluster[3]. As a result of that crawl, 39,024 users were crawled and 770,595 users were scheduled giving an average of 4.09 users per second (39,024 users in 159 minutes). Also, in that time, 372,931 unique interests were discovered and users declared a total of 2,151,090 interests, giving a mean of 55.12 interests per user and a mean of

[3] Information on Beocat can be found at http://support.cis.ksu.edu/BeocatDocs

5.77 unique interests per user. From the users crawled, 2,992,607 directed links were found, giving a mean of 76.69 links per user.

3.2 Base Feature Selection

Base feature selection is an essential part of the overall approach. Without a broad set of base features, the constructed features may not adequately represent the search space. If the constructed features under represent the problem space then the classifier's performance will suffer as a result. As discussed in Chapter 1, base features are typically gathered from a domain expert and from among the available data. [Hsu06], [Hsu07] and [Dam08] all study a similar link mining problem and use a similar feature set. This feature set includes features that are (1) user-dependent, (2) pair-dependent, and (3) graph-dependent. From among the available features I chose the features that I believe best captured the breadth of the problem. They are discussed in the following subsections.

3.2.1 User-dependent Base Features

User dependent features are those features that are based solely on a single individual. For example, the age of a person is a user-dependent features, whereas the difference ages is a pair-dependent feature. There are relatively few user-dependent features because I believe that they do not sufficiently address of link mining problem. Nevertheless, I include

1. Number of interests listed by u.
2. Number of interests listed by v.

in the list of all features.

3.2.2 Pair-dependent Base Features

Pair dependent features are those features that are based on a pair of individuals. For example, the size of the intersection of each individual's listed interests is a pair-dependent feature, whereas the size of an individual's interest list alone would be considered a user-dependent feature. Pair dependent features are very important to consider when performing link mining. Intuitively, the greater the intersection of interests the more likely the individuals are alike and therefore are friends. Inversely, if a pair has a large difference in interests the more

likely that they have little in common and are consequently not friends. For this study I include features based on

1. Interests of u and v.
2. Friends of u and v.
3. Location of u and v.

in the list of all features.

3.2.3 Graph Base Features

Graph features are those features that are based on the relative locations of a pair of individuals. For example, if u is a friend of v then u and v are likely to be very close to one another in the graph, not considering the actual friendship link between u and v. Inversely, if u and v are not friends and do not share any friends then they will likely be very far apart in the graph. For this study I include graph features based on

1. The indegree of u: the popularity of u
2. The indegree of v: the popularity of v
3. The outdegree of u: the number of friends of u
4. The outdegree of v: the number of friends of v
5. Shortest path length between u and v
6. Shortest path length between v and u: backward distance

in the list of all features.

This list of features is by no means exhaustive. Rather it is a sample of features from other works that I find useful for this particular problem. More features have been crawled and gathered, but in the interest of brevity and maintaining a reasonable scope they are omitted.

3.3 Candidate Pair Generation

Generating candidate pairs for link mining operations is an important task. Because the ultimate goal is to predict the existence of a link between two nodes a method for generating a candidate pair $\langle u, v \rangle$ is needed. Randomly picking two nodes to form a pair initially seems to be a sound method, however Subsection 3.3.1 shows that the graph density of the corpus is too

sparse to give testable pairs. Therefore a less random method for generating candidate pairs is necessary.

3.3.1 Graph Density

Graph density is the number of edges relative to the number of vertices. For instance, a dense graph is a graph in which the total number of edges is close to the maximal number of edges. Symmetrically, the sparse graph has a relatively small number of edges. Chartrand shows that the maximal number of edges $K_n$where $n = |V|$ is $K_n = \frac{n(n-1)}{2}$. [Cha85] By this definition, a graph with n vertices is sparse if $|E| \ll K_n$ and dense otherwise.

By applying this definition to the crawl results in Subsection 3.1.1 it is seen that $n = 39{,}024$ and therefore $K_n = 2.96 \times 10^{11}$. Because $|E| = 2{,}992{,}607$, which is sufficiently less than K_n, the graph used in this thesis is considered sparse.

Sparse graphs such as the social networks graph used in this work do provide poor candidate-pair generation because pairs are selected at random. This is because the odds of randomly generating a pair that is connected is $\frac{|E|}{K_n} = 0.00393$. Therefore, this method would need to generate about 300 negative pairs for every 1 positive pair. The poor odds of generating a positive candidate pair would lead to inaccurate and unstable results. [Kub97]

3.3.2 Forced Parity Candidate Generation

In order for the learning model to be accurate, the training set must contain a sufficient number of both positive and negative examples. While the ratio of positive examples to negative examples does not need to necessarily be 1, a sufficient number of both should exist. [Kub97] To resolve the problem that a random approach presents I force the example set to have 50% positive examples and 50% negative examples.

Caution should be observed when transforming the distribution of the example set. Performing this redistribution on the training data given to the machine learning algorithm may provide an inaccurate representation of the original distribution, therefore if the learned classifier is tested against data from the original distribution the traditional accuracy, precision and recall

metrics will not accurately describe the validity of the classifier. For example, consider a training set with 1000 total examples where 950 examples are negative and only 50 are positive. A poorly trained classifier might simply assign *no* to all of the examples and would still receive a cross validation accuracy of 95% (950/1000). These results are misleading because a poorly-trained classifier can still give good results. Subsection 3.4.1, therefore, outlines metrics that are better suited for judging the performance of classifiers trained with imbalanced data.

3.4 Genetic Feature Construction

Evolutionary algorithms can be used to change the representation of the inputs for machine learners. In this work I use a widely known genetic programming technique called *symbolic regression* which was described in Subsection 1.2.3, as well as an unexplored approach which uses multiple symbolic regression trees to construct multiple features.

3.4.1 Symbolic Regression

In the symbolic regression approach, an optimal prediction function can be obtained by means of a genetic program. With this approach, a single mapping function can be learned by constructing a symbol tree with a single base feature as the leaves. For example, in a training set with examples containing 7 features, including a class feature (*yes*/*no*), a single tree would be learned and applied to each of the 7 features. In this way, a feature set from the original dataset would be transformed into a feature set where all of the features have been mapped with the regressed function. Table 3.1 shows this example where 2 instances with their 7 features are mapped to a new set of instances from the symbol tree: `(% (* (% x x) (* x x) ) (+ x x))`, which is in prefix-notation and where `%` represents protected division (not the modulus operator).

Table 3.1: Example Symbolic Regression Interpretation

Features	ID of u	ID of v	Forward Deleted Distance	Intersection of Interests	Intersection of Friends	Outdegree of u	Friends
Instance 1	150	254	3	15	63	151	yes
Instance 2	64	125	6	19	14	25	no
Symbol Tree	`(% (* (% x x) (* x x) ) (+ x x))`						n/a
Instance 1 Result	75	127.0	1.5	7.5	31.5	75.5	yes
Instance 2 Result	32	62.5	3	9.5	7	12.5	no

This symbol tree can be represented graphically, as in Chapter 1, by the tree shown in Figure 3.3.

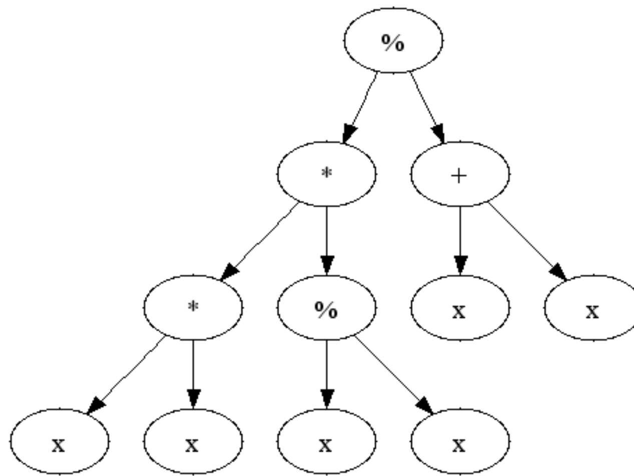

Figure 3.3: Example symbol tree used in Table 3.1

As Figure 3.3 shows, the simple regression tree approach is limited because it attempts to apply the same symbol tree on distinct features. This is problematic because different features ought to be optimized individually. Subsection 3.4.2 is formulated to solve this specific shortcoming.

3.4.2 Feature Construction via Multiple Regression Trees

To optimize individual symbol trees per feature, 10 regression trees were genetically computed on the 108 base features. Afterwards, the newly constructed features were used to train

and cross validate a classifier. The major difference between this method and that of a single symbol tree is that the symbol trees' leaves represent different features rather than a having a single tree applied to all features. For example, consider the features and trees from Table 3.2 where each of the 10 features are constructed solely from a symbol trees with base features as the leaves. Note that unlike in the previous section, the features in Table 3.2 are distinct. That is, `F1` represents forward deleted distance, and `F7` represents Euclidean distance, etc.

Table 3.2: Example of features constructed with multiple regression trees

Constructed Feature	Symbol Tree
1	`(* F1 F7)`
2	`(* (% F9 F54) )`
3	`(+ (* F6 F8) (- F67 F87)`
4	`F2`
5	`(+ F15 F16)`
6	`(- (* F4 F6) F46)`
7	`F5`
8	`(% (% F45 F39) (* F70 F3)`
9	`(- F7 F91)`
10	`(* F1 F1)`

In many cases the constructed features might not make any logical sense and an inappropriate conclusion may be drawn from the intricate details of the specific sampling. Therefore, I ran this experiment several times, each time with a different seed. Finally, I examined the commonalities from among the different executions in order to draw appropriate conclusions about the data set.

In all symbolic regression experiments, only four symbols are used: addition, subtraction, multiplication and protected division (where a division by 0 returns 1). [Koz92]

3.5 Feature Computation

After the genetic program constructs new features, their values need to be computed before any learning can begin. Set computation is algorithmically easy, but its theoretical

complexity can be problematic for sufficiently large sets. For example, if the genetic program constructs a feature composed of the intersection of u's interest set I_u with the interest set I_v of v then the computation time is $\theta(|I_u| \times |I_v|)$, which is expected to be $\theta(n^2)$ in the expected case where n is the average number of user interests. Section 3.1 shows that the average number of interests per user is about 55. Although $55^2 = 3{,}025$ comparisons in the expected case are not of too much concern, it is important if scalability is considered. Set union and difference operations are performed in a similar fashion and can be shown to have the same computational complexity.

Set computation is shown above to not be of much concern because of the relatively small size of the sets. However, computation of the graph features is a difficult problem because of the relatively large size of the graph. Consider the crawled graph with 770,595 vertices and 2,992,607 edges, in order to compute the distance between u and v a shortest path algorithm is needed. Currently, the fastest shortest path algorithm is Dijkstra's Shortest Path Algorithm which operates in $O(|\mathrm{E}| + |V|^2)$, or simply $O(|V|^2)$. Because our graph's edges are all of equal weight/distance a small adaptation on Dijkstra's algorithm becomes Breadth First Search (BFS) which operates in $O(|\mathrm{E}| + |V|)$. The computational complexity of even the fastest known algorithm is unacceptable for such a large graph. Therefore I propose the following approximation to find the shortest path between two nodes in a graph.

3.5.1 Shortest Path Approximation

To find the shortest path in a very large graph I considered how humans are able to quickly approximate the shortest path between two points on a street map. For instance, when humans look at a street map we are able to very quickly determine the general direction from start to destination (*e.g.*, left is East, up is North). Without any background information concerning speed limits or traffic conditions, we pick a route that provides the clearest path towards the destination. I hypothesize that by treating a graph as a street map and computationally approximating path decisions, I will be able to quickly find an approximate shortest path between two arbitrarily connected vertices. To do this, I need to create an accurate Cartesian embedding of the graph, and then I will be able to navigate the graph according to the Cartesian directions and distances gained from the embedding.

3.5.1.1 Graph Embedding

Graph embedding, otherwise known as graph drawing, is the standard means for the visualization of relational information. Its usefulness depends on the readability of the resulting layout; that is, the drawing algorithm's ability to convey the meaning of the graph. Many graph drawing algorithms have been developed [DiB99] [Kau01]. However, these approaches can be prohibitively slow for sufficiently large graphs. Harel and Koren observed that laying out a graph in a high dimension is much easier to do than drawing it in 2-dimensions [Har02]. Therefore they propose an approach that initially lays out the graph in a very high dimensional space (*e.g.*, in 50-100 dimensions) and then projecting that layout into a more visually appealing 2-dimensions. To do this the authors use principal components analysis (PCA) which is described in Subsection 1.1.1.

The result is an extremely fast, simple algorithm that was able to draw the 770,595-vertex graph in about 3 hours. By laying out the graph I was able to ascertain 2-dimensional Cartesian points. These points were stored in the `VERTICES` table in the database.

3.5.1.2 Finding the Approximate Shortest Path

Once Cartesian coordinates are assigned to vertices the shortest path problem becomes a matter of directed search. Therefore I adopted the popular A*-search algorithm to suit this problem. This approach allows me to specify three cost functions: $g(x)$, which is the actual shortest path distance traveled from the initial vertex to the current vertex; $h(x)$, which is the estimated or *heuristic* distance from the current node to the goal; and $f(x)$, which is the sum of $g(x)$ and $h(x)$, *i.e.*, the heuristic shortest path distance. To find $g(x)$ I need to simply keep track of the distance that I have searched, and because the edges are un-weighted this is simply the sum of the Cartesian distances between all vertices in the current path. To find $h(x)$ I look at the straight-line distance between the current vertex and the destination vertex. This is possible because the algorithm is given the location of the destination vertex. For more general search problems where the location of the destination is not known this approach would not work. By knowing $g(x)$ and $h(x)$ I can keep an open-set of candidate vertices ranked by $f(x)$ and picking paths based on the candidates with the lowest $f(x)$ score until the destination is reached.

By applying these two approaches I was able to find an accurate Cartesian layout of the graph with a one-time computation time of about 3 hours. By using the Cartesian coordinates gleaned from the layout I was able to use an A*-search with a Cartesian distance heuristic to quickly find the shortest path between two vertices[4].

3.6 Experiment Design

This thesis aims to test the ability of a genetic program to construct features which improve the accuracy of current link mining algorithms. In order to do so, different sets of experiments were performed on identical training data, namely the social networks data described in Section 3.1.

First, as a baseline, the original features were used to train and cross-validate three types of learning algorithms: J48, which is a tree learning algorithm based on ID3, NaïveBayes, which is a probabilistic learning algorithm which uses Bayes' theories of probability, and Logistic, which is a function finding method similar to symbolic regression, and OneR, which simply picks the best possible *single* rule. [Wit99]

Second, the genetic program was invoked to learn a classifier based on the application of a single symbolic regression tree as described in Subsection 3.4.1. The fitness function of the genetic program was determined based on the area under the receiver operating characteristic (AUC) metric of the classifier. Because of the random nature of evolutionary algorithms, the symbolic regression experiment was run 5 times.

Third, the genetic program was invoked to learn a classifier based on the application of multiple symbolic regression trees as described in Subsection 3.4.2. The fitness function of the genetic program was determined by the AUC metric of the classifier. Because of the random nature of evolutionary algorithms, the multiple symbolic regression experiment was run 5 times for each classifier. Appendix B contains the ECJ parameter file used in the configuration of the genetic program. Because the test hold out data was used to reinforce the genetic program, it can

[4] Early analytical results show that for a graph size of 50,000 the mean time to find the shortest path between random vertices is 5.44 milliseconds with the approach described in this section compared to a mean time of 229.88 milliseconds for Dijsktra's algorithm. Although this is an approximation the algorithm generally achieves 100% accuracy.

be argued that the test hold out data is an inconclusive indicator of the algorithm performance. Therefore, validation holdout data with 2000 examples of the original distribution was used to validate the performance of the various learning algorithms.

Finally, the genetic program performance results were compared to those of the meta-learning algorithms such as bagging, boosting, etc. described in Section 2.1.

3.6.1 Evaluation Metrics

The Receiver Operating Characteristic (ROC) curve is an alternative to accuracy for the evaluation of classifiers. The ROC curve is a curve and not a single number statistic. In particular, this means that the comparison of two algorithms on a dataset does not always produce an obvious order. Accuracy, on the other hand, is 1 – error rate, and is the standard method used to evaluate learning algorithms. Accuracy is a single-number evaluation of performance. [Pro05]

According to Wikipedia [Wik08]:

> A ROC space is defined by the false positive rate (FPR) and the true positive rate (TPR) as x and y axes respectively, which depicts relative trade-offs between true positive (benefits) and false positive (costs). Since TPR is equivalent with sensitivity and FPR is equal to 1 - specificity, the ROC graph is sometimes called the sensitivity vs (1 - specificity) plot. Each prediction result or one instance of a confusion matrix represents one point in the ROC space.
>
> The best possible prediction method would yield a point in the upper left corner or coordinate (0,1) of the ROC space, representing 100% sensitivity (no false negatives) and 100% specificity (no false positives). The (0,1) point is also called a *perfect classification*. A completely random guess would give a point along a diagonal line (the so-called *line of no-discrimination*) from the left bottom to the top right corners. An intuitive example of random guessing is a decision by flipping coins (head or tail).

The area under the ROC curve (AUC) is a single-number statistic often used to rank ROC curves. More importantly, for these experiments the AUC can handle the problems that were

described at the end of Section 3.4 regarding accuracy, precision and recall statistics. Most results in this thesis will be evaluated using the Area under the ROC Curve (or AUC).

CHAPTER 4 - Results

This chapter presents four sets of results for each of the four experiments described in Section 3.6. The standard metrics used will often be accompanied by a more descriptive analysis of the results. The final section in this chapter will view the results in a comparative paradigm setting up the final chapter containing the conclusions that can be drawn from these experiments. The implementations of all algorithms are from WEKA 3.5.7[5].

4.1 Results from Traditional Learning using Base Features

This section looks at the performance of traditional machine learning techniques using the base features. For these experiments the 2000 instances were calculated each with 108 features annotated in Appendix 1. The training data has 50% positive examples and 50% negative examples.

As described in Chapter 1, many learning algorithms, especially tree algorithms such as J48, C4.5, etc., use entropy as the means for ranking and dividing attributes. Therefore, it is

[5] WEKA 3.5.7 is available at http://www.cs.waikato.ac.nz/ml/weka/

helpful to look at the ranking of the attributes in terms of information gain (*i.e.*, entropy). Table 4.1 shows that `intersectionfriendsidcount` (the count of the intersection of friends) is the highest ranking feature followed by other statistics based on the intersection of friends followed by `fdd` (forward deleted distance) and `bdd` (backward deleted distance. These findings differ slightly from previous research in which `bdd` and `fdd` were consistently the features with the highest entropy.

Table 4.1: Attribute Ranking in terms of Information Gain **[Mit97]**

Ranking	ID	Feature Name
0.39492	22	`intersectionfriendsidcount`
0.39492	27	`intersectionfriendsagecount`
0.38747	18	`intersectionfriendsidsum`
0.38298	19	`intersectionfriendsidavg`
0.26334	83	`firstfriendsagesum`
0.24935	23	`intersectionfriendsagesum`
0.24732	24	`intersectionfriendsageavg`
0.22317	1	`fdd`
0.20616	2	`bdd`
0.20546	3	`indegreeu`
0.17388	43	`unionfriendsagesum`
0.15208	84	`firstfriendsageavg`
0.14285	79	`firstfriendsidavg`
0.13935	4	`outdegreeu`
0.13935	82	`firstfriendsidcount`
0.13935	87	`firstfriendsagecount`
0.11199	71	`firstinterestsidmax`
0.09812	69	`firstinterestsidavg`
0.08678	44	`unionfriendsageavg`
0.08443	15	`intersectioninterestspopularitymin`
0.07647	78	`firstfriendsidsum`
0.07029	39	`unionfriendsidavg`
0.06293	7	`euclidDist`
0.06171	8	`intersectioninterestsidsum`

Next, the same data was used to train a classifier using several traditional learning algorithms. In each experiment the classifier was trained on the 2000 instances and a holdout

data set with a 1.5% positive example rate (roughly matching that of the original distribution) was applied to test the performance of the classifier. For the purposes of this research, the learning algorithm which produces a classifier that maximizes area under the ROC (AUC) was declared the winner. Table 4.2 presents results from the various learning algorithms including the AUC, confusion matrix breakdown and the accuracy, precision and recall statistics. Other performance metrics can be calculated based on the confusion matrix breakdown as needed, but are not needed for this work. Note that the precision and recall statistics give two percentages: the first for the positively labeled instances, and the second for the negatively labeled instances.

Table 4.2: Results for classifiers trained on base features, tested with 10-fold cross validation.

Learning algorithm	AUC	Confusion Matrix		Accuracy	Precision	Recall
OneR	82.7	703	298	82.65	93.5/76.1	70.2/95.1
		49	950			
J48	89.5	903	98	90.4	90.6/90.2	90.2/90.6
		94	905			
IB1	74.3	695	306	74.25	76.9/72.1	69.4/79.1
		209	790			
Logistic	95.4	891	110	91.2	93.1/89.5	89/93.4
		66	933			
NaiveBayes	92.1	786	215	85.1	90.4/81	78.5/91.7
		83	916			

These results correspond with results from earlier experiments on smaller data sets wherein the *J48* learning algorithm performed the best at maximizing the accuracy. However, in these results the *Logistic* learning algorithm outperformed all other algorithms. Again, these results are not conclusive because testing *must* be done on examples from the original distribution.

In order to more deeply understand the results of the J48 learning algorithm it helps to visualize the generated decision tree. This decision tree shown in Figure 4.1 has 59 leaves and 117 total nodes. The distance from the root of each decision node largely coincides with the information gain rankings from Table 4.1. That is, the higher the information gain ranking, the higher the corresponding node is in the decision tree.

Figure 4.1: Decision tree for J48 classifier trained on base features.

Judging from the size of the decision tree (58 decision points) relative to the total number of features, the induced classifier does not seem to overfit, but in order to empirically test that assumption, the same classifier is tested against data that it did not train on. This test data again contains 2000 instances with 29 positive and 1,971 negative examples or about 1.5% positive examples.

Table 4.3 shows the results of training and testing on two independent data sets (*i.e.*, no cross validation). The performance does not decrease, and in some cases the classification performance actually increases. These results show that the classifiers being constructed do not overfit and are good representatives for the larger distribution.

Table 4.3: Results for classifiers trained on base features, tested with hold out data with original distribution.

<table>
<tr><th>Learning algorithm</th><th>AUC</th><th colspan="2">Confusion Matrix</th><th>Accuracy</th><th>Precision</th><th>Recall</th></tr>
<tr><td rowspan="2">OneR</td><td rowspan="2">86.7</td><td>23</td><td>6</td><td rowspan="2">93.95</td><td rowspan="2">16.7/99.7</td><td rowspan="2">79.3/94.2</td></tr>
<tr><td>115</td><td>1856</td></tr>
<tr><td rowspan="2">J48</td><td rowspan="2">88.7</td><td>27</td><td>2</td><td rowspan="2">90.05</td><td rowspan="2">12.1/99.9</td><td rowspan="2">93.1/90.0</td></tr>
<tr><td>197</td><td>1774</td></tr>
<tr><td rowspan="2">IB1</td><td rowspan="2">77.4</td><td>21</td><td>8</td><td rowspan="2">82.15</td><td rowspan="2">5.7/99.5</td><td rowspan="2">72.4/82.3</td></tr>
<tr><td>349</td><td>1622</td></tr>
<tr><td rowspan="2">Logistic</td><td rowspan="2">98.0</td><td>28</td><td>1</td><td rowspan="2">94.05</td><td rowspan="2">19.2/99.9</td><td rowspan="2">96.6/94.0</td></tr>
<tr><td>118</td><td>1853</td></tr>
<tr><td rowspan="2">NaiveBayes</td><td rowspan="2">91.4</td><td>22</td><td>7</td><td rowspan="2">91.7</td><td rowspan="2">12.2/99.6</td><td rowspan="2">75.9/91.9</td></tr>
<tr><td>159</td><td>1812</td></tr>
</table>

As a final test of performance, ROC curves are generated in order to visualize the overall performance of the learning algorithms. Figure 4.2 shows the ROCs for the various learning algorithms (IB1 could not generate a ROC). Logistic performed best with an AUC of 98%.

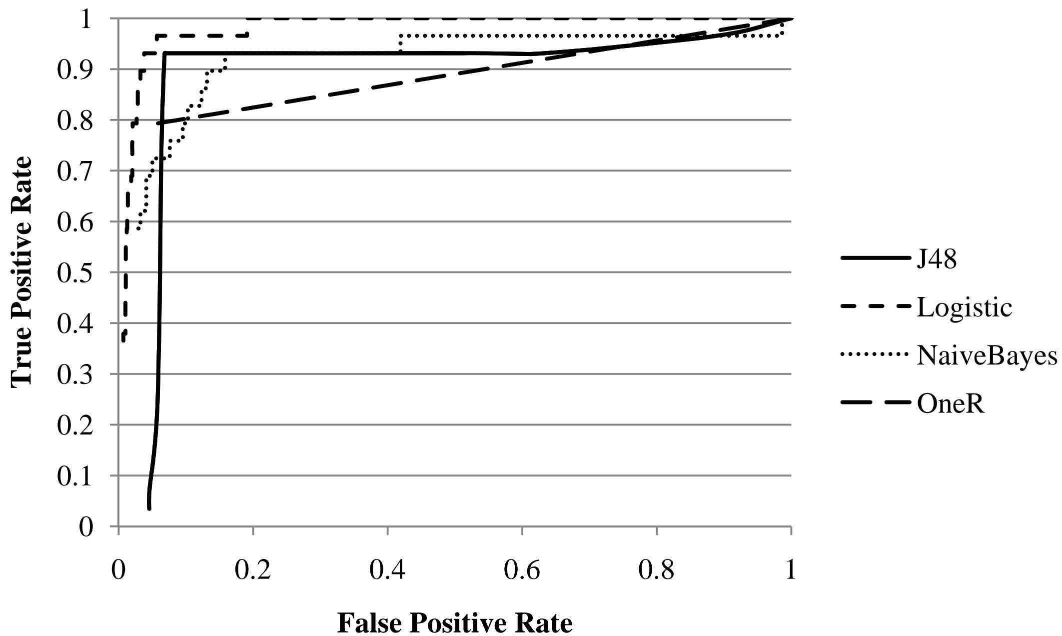

Figure 4.2: ROC curves for learning algorithms trained and cross-validated

4.2 Results of Feature Construction using a Single Symbol Tree

In an initial test of the genetic programming approach to feature construction, a symbolic regression tree was genetically grown according to the specifications in Subsection 3.4.1. Because this was an initial test of the capabilities of the genetic program, the results in this section are much less robust than in other sections. Nevertheless, these results are reported to highlight the shortcomings in a simple symbolic regression approach.

Figure 4.3 shows the result of applying a single symbol tree to each of the features. The fitness is in terms of AUC (87.129% in this result) of a J48 classifier. As the flat fitness curve shows, the results do not improve regardless of the symbol tree that is used to modify the feature values.

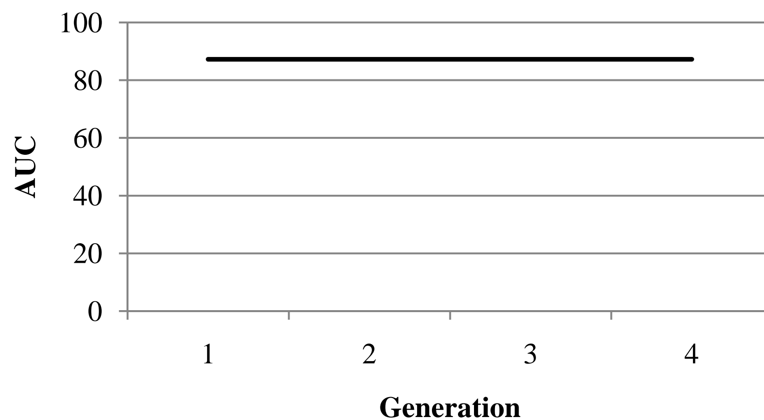

Figure 4.3: Progression of the fitness of the best individual through 4 generations. Higher is better.

Table 4.5 shows the symbol trees which were generated by the genetic program and the corresponding fitness of the symbol trees. The reasons for these results are discussed as a conclusion in Chapter 5.

Table 4.4: Symbol trees and corresponding fitness for 4 generations of single symbolic regression

Generation	Symbol Tree	Fitness
1	`(* x x)`	87.129
2	`(sin (- (exp (% x x)) (% (exp x) (sin x))))`	87.129
3	`(rlog (- (exp (- (exp (* (cos x) (% x x))) (+ (sin (sin x)) (cos (+ x x))))) (+ (sin (sin x)) (cos (+ x x)))))`	87.129
4	`(sin (- (exp (% x x)) (% (exp x) (sin x))))`	87.129

4.3 Results of Feature Construction using Multiple Symbol Trees

The results of the major contributions of this work are presented in this section. I first present the results of several iterations of feature construction experiments. In each experiment the total number of constructed features was exactly 10, and the genetic program's population was exactly 100 individuals. As explained in the previous chapter, the learning algorithm was trained with 2000 examples of a 50/50 distribution and tested on an independent set of 2000 examples of the original distribution; because of this "wrapper" approach, the test examples influenced the learning algorithm, therefore another independent validation set of 2000 examples of the original distribution was used to rate the final performance of each algorithm. The scores reported in this section are from the holdout validation data.

4.3.1 OneR

First, the genetic program was executed using the OneR learning algorithm in order to train a classifier which was then tested on test hold out data of the original 1.5%-positive distribution, results are computed by validating the resulting classifier on validation hold out data of the original 1.5%-positive distribution. 1 minus the AUC of the classifier was used as the fitness function. In total, five independent executions were tried each with a random starting seed. Figure 4.4 shows the fitness of the best individual from each generation of these executions, wherein "Base" is the non-GP result from Section 4.1, and "Avg" is the mean of the five runs. The "Avg" fitness (1 – AUC) remains relatively consistent at 13.16%.

As Figure 4.4 illustrates, the genetic program was unable to genetically construct features to improve the AUC over the five generations. Exploratory repetitions show that the results

remain steady up to 50 generations.

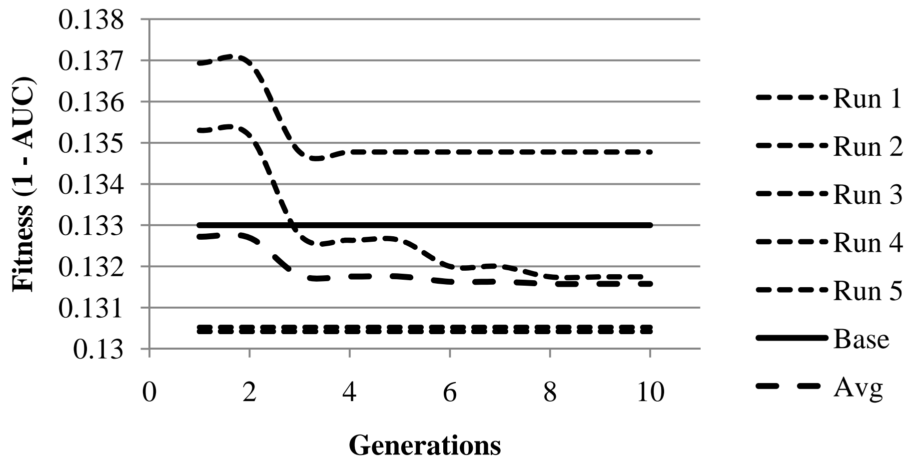

Figure 4.4: Best individuals per generation with OneR for 5 GP runs. Lower is better.

Despite these flat results the population does become more specialized as shown in Figure 4.5, where the mean validation score over the *entire* population increases (error decreases). Specifically, the "Avg" AUC begins at 27.88% in the first generation and decreases to 13.37% in the tenth generation.

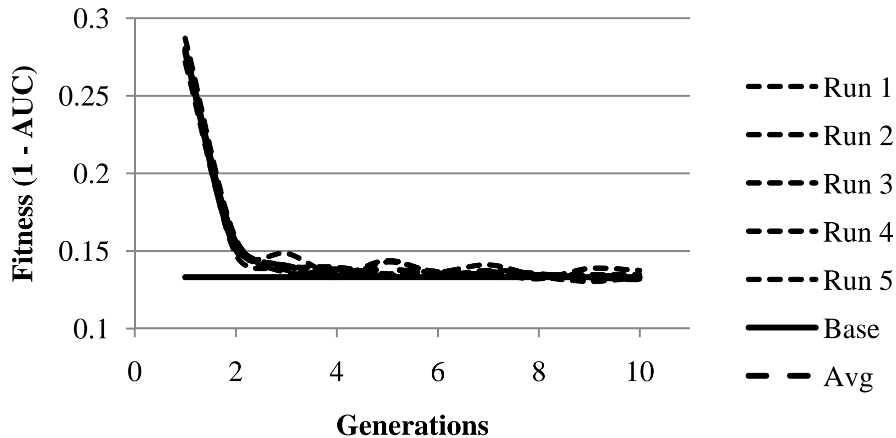

Figure 4.5: Average validation score over the entire population per generation with OneR for 5 GP runs. Lower is better.

4.3.2 Logistic

In the same manner as above, the genetic program was executed again using the Logistic learning algorithm in order to train a classifier which was tested on test hold out data of the original 1.5%-positive distribution, results are computed by validating the resulting classifier on validation holdout data of the original 1.5%-positive distribution.

1 minus the AUC from the classifier was used as the fitness function. In total, five independent executions were tried each with a random starting seed. Figure 4.6 shows the fitness of the best individual from each generation of these executions, where "Base" is the non-GP result from Section 4.1, and "Avg" is the mean of the five runs. The "Avg" validation score (1 – AUC) begins at 2.5% in the first generation and decreases to 1.77% in the tenth generation.

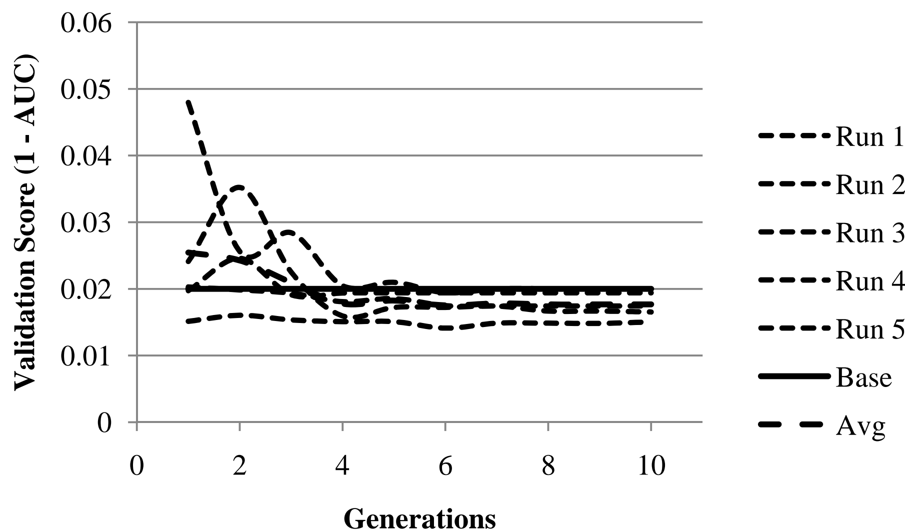

Figure 4.6: Best individuals per generation with Logistic for 5 GP runs. Lower is better.

The results from the experiment using the Logistic learning algorithm show a 0.78% average decrease in the validation score of the classifier (*i.e.*, 0.78% increase in AUC). Furthermore, the mean validation score of the entire population decreases dramatically from 17.80% in the first generation to 1.77% in the tenth generation as shown in Figure 4.7.

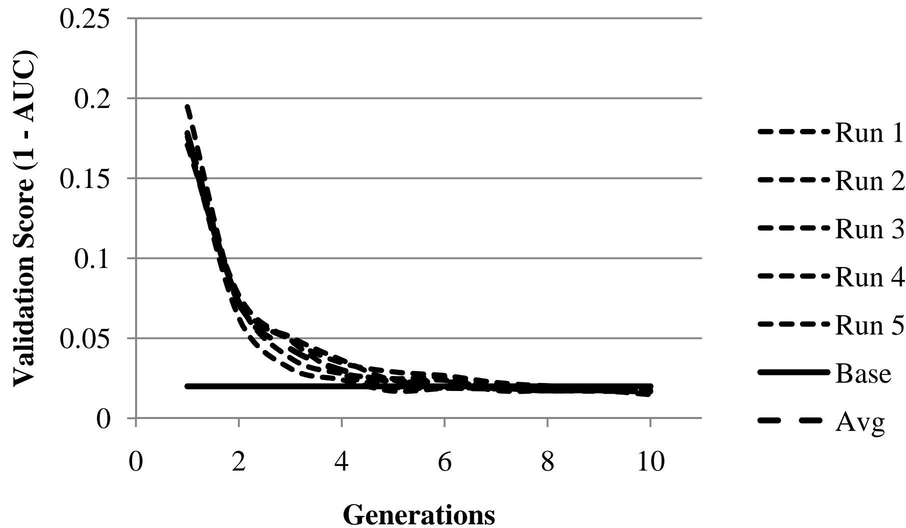

Figure 4.7: Average validation score over the entire population per generation with Logistic for 5 GP runs. Lower is better.

4.3.3 J48

Again, the genetic program was executed using the J48 decision tree learning algorithm in order to train a classifier which was tested on test hold out data of the original 1.5%-positive distribution, results are computed by validating the resulting classifier on validation hold out data of the original 1.5%-positive distribution.

1 minus the AUC from the classifier was used as the fitness function. In total, five independent executions were tried each with a random starting seed. Figure 4.8 shows the validation score of the best individual from each generation of these executions, where “Base” is the non-GP result from Section 4.1, and “Avg” is the mean of the five runs. The “Avg” validation score (1 – AUC) begins at 3.69% in the first generation and decreases to 2.21% in the tenth generation.

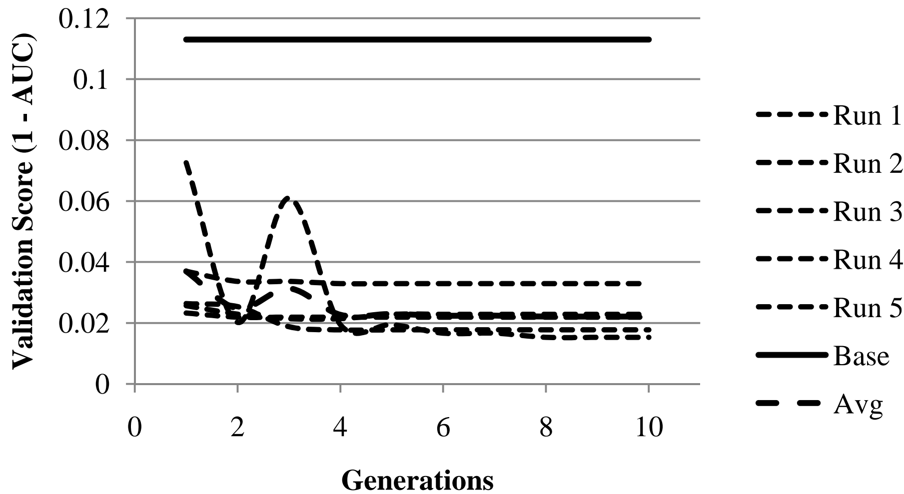

Figure 4.8: Best individuals per generation with J48 for 5 GP runs. Lower is better.

The results from the experiment using the J48 learning algorithm show a 1.48% average decrease in the validation score of the classifier (*i.e.*, 1.48% increase in AUC). Furthermore, the mean validation score of the entire population decreases dramatically from 17.34% in the first generation to 2.23% in the tenth generation as shown in Figure 4.9.

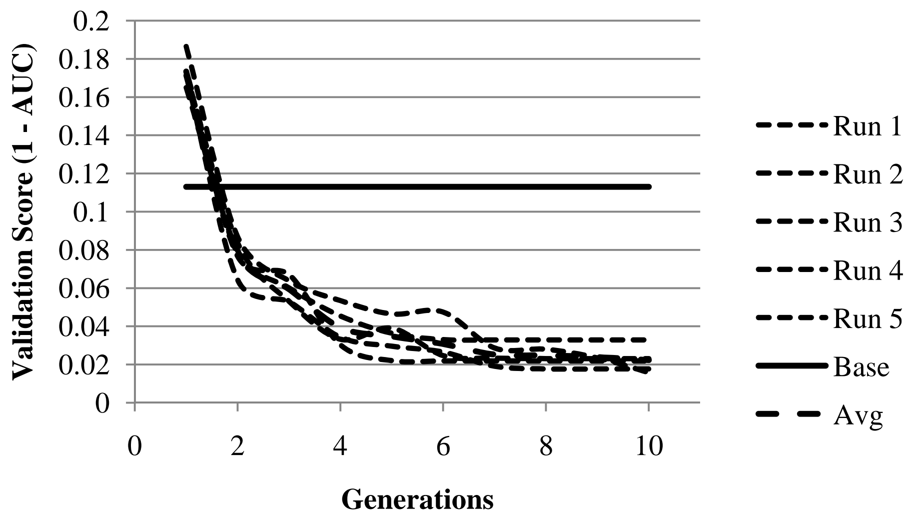

Figure 4.9: Average validation score over the entire population per generation with J48 for 5 GP runs. Lower is better.

4.3.4 NaiveBayes

Again, the genetic program was executed using the NaiveBayes learning algorithm in order to train a classifier which was tested on test hold out data of the original 1.5%-positive distribution, results are computed by validating the resulting classifier on validation hold out data of the original 1.5%-positive distribution.

1 minus the AUC from the classifier was used as the fitness function. In total, five independent executions were tried each with a random starting seed. Figure 4.10 shows the validation score of the best individual from each generation of these executions, where "Base" is the non-GP result from Section 4.1, and "Avg" is the mean of the five runs. The "Avg" validation score (1 – AUC) begins at 3.32% in the first generation and decreases to 2.40% in the tenth generation.

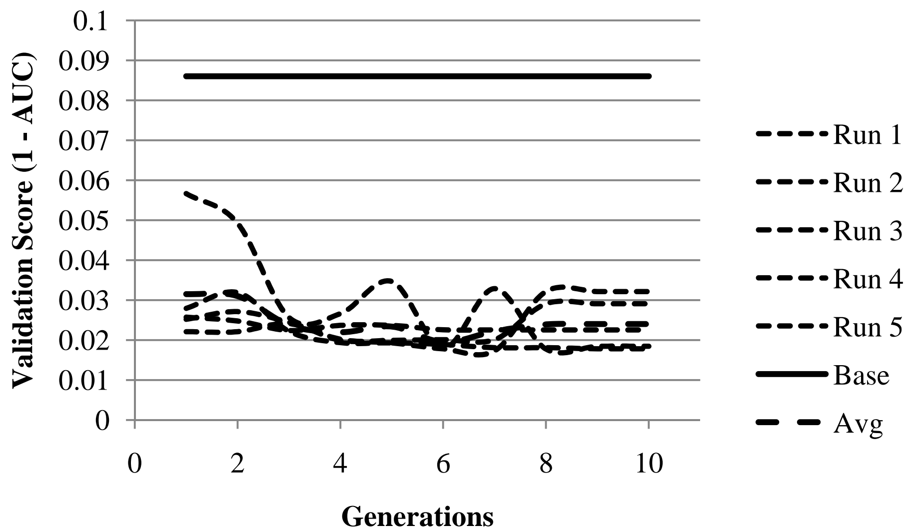

Figure 4.10: Best individuals per generation with NaiveBayes for 5 GP runs. Lower is better.

The results from the experiment using the NaiveBayes learning algorithm show a 0.75% average decrease in the validation score of the classifier (*i.e.*, 0.75% increase in AUC). Furthermore, the mean validation score of the entire population decreases dramatically from 22.66% in the first generation to 2.34% in the tenth generation as shown in Figure 4.11.

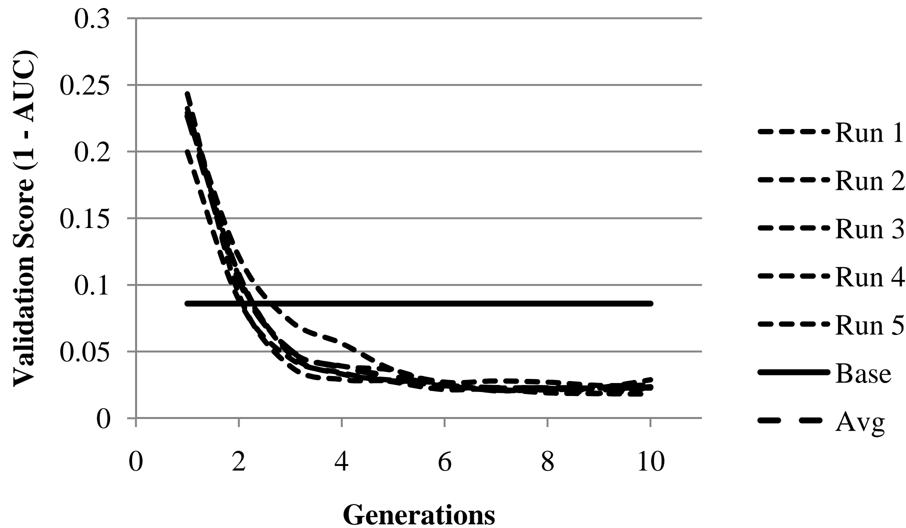

Figure 4.11: Average validation score over the entire population per generation with NaiveBayes for 5 GP runs. Lower is better.

4.3.5 IB1

Again, the genetic program was executed using the IB1 learning algorithm in order to train a classifier which was tested on test hold out data of the original 1.5%-positive distribution, results are computed by validating the resulting classifier on validation hold out data of the original 1.5%-positive distribution.

1 minus the AUC from the classifier was used as the fitness function. In total, five independent executions were tried each with a random starting seed. Figure 4.10 shows the validation score of the best individual from each generation of these executions, where "Base" is the non-GP result from Section 4.1, and "Avg" is the mean of the five runs. The "Avg" validation score (1 – AUC) begins at 13.20% in the first generation and decreases to 8.42% in the tenth generation.

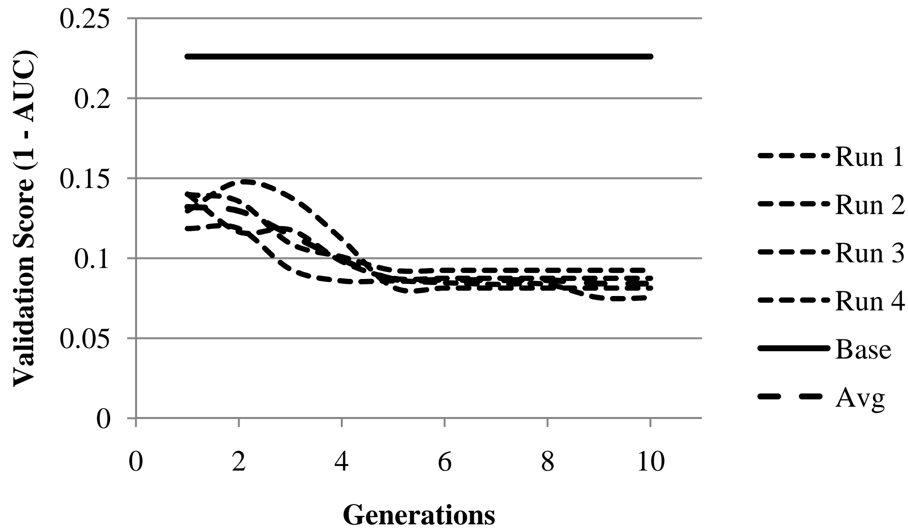

Figure 4.12: Best individuals per generation with IB1 for 4 GP runs. Lower is better.

The results from the experiment using the IB1 learning algorithm show a 4.79% average decrease in the validation score of the classifier (*i.e.*, 4.79% increase in AUC). Furthermore, the mean validation score of the entire population decreases dramatically from 30.94% in the first generation to 8.62% in the tenth generation as shown in Figure 4.13.

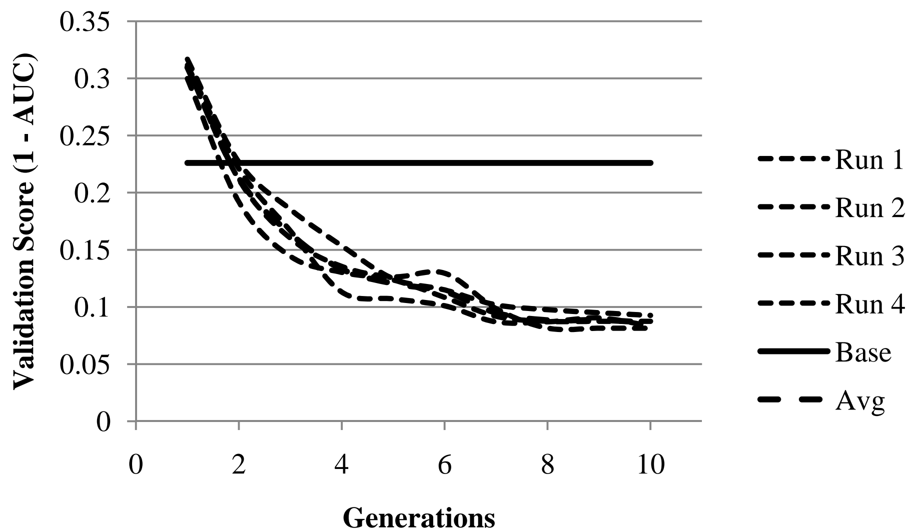

Figure 4.13: Average validation score over the entire population per generation with IB1 for 4 GP runs. Lower is better.

4.3.6 Comparative Analysis

Now that the individual learning algorithms' results have been presented in Subsections 4.3.1 through 4.3.5, a comparative analysis can be performed on the averages from each learning algorithm's results. Figure 4.14 shows the average validation scores for the best individuals for all generations. Note all experiments ran for an equal number of generations, and each run had converged after by at least the tenth generation. A single execution was deemed to have a sufficient likelihood of having converged when the best individual's fitness (*i.e.*, 1 – AUC) was identical three generations in a row.

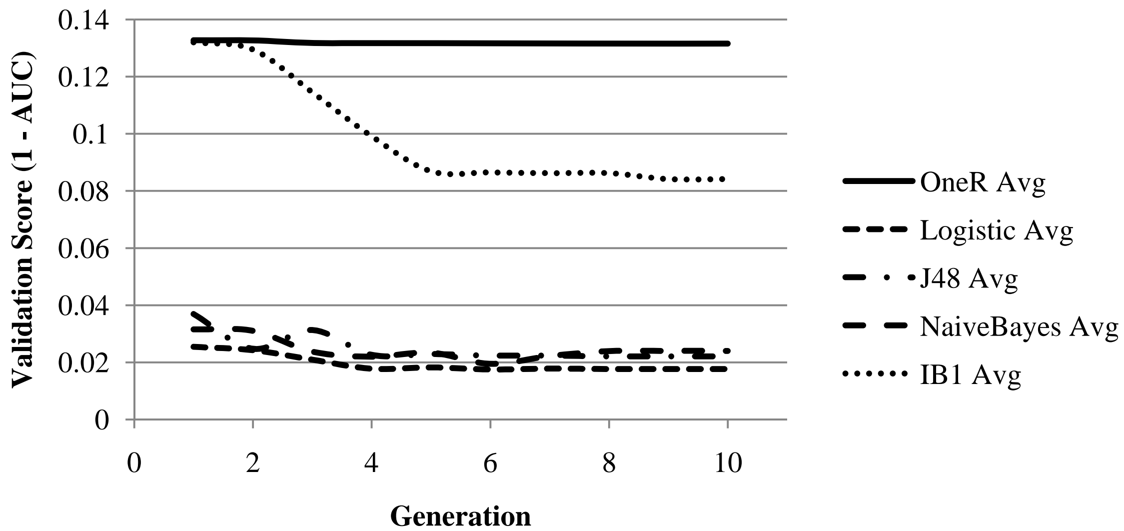

Figure 4.14: Average validation scores for the best individuals for all generations. Lower is better.

With one exception in OneR, all results effectively converged to an relative optima in which the fitness was lower than that of the base classifier. Depending on how Figure 4.14 is interpreted one algorithm might be said to outperform another. The absolute performance of the specific learning algorithm is not important to the outcome of this thesis; rather, the success or failure lies in the ability of the genetic program to construct features that, if provided to *any* learning algorithm will decrease the error (*i.e.*, increase the fitness).

Table 4.5: Classifier results in terms of AUC (1 – fitness) compared to base results. Higher is better.

Learning Algorithm	GP-Results	Base-Results	Increase
OneR	86.96	86.7	0.26
Logistic	98.50	98.0	0.50
J48	98.47	88.7	9.77
NaiveBayes	98.22	91.4	6.82
IB1	92.46	77.4	15.06

4.3.7 Constructed Feature Trees

In order to gain a better understanding of the general underlying structure of the features that are being genetically constructed this subsection enumerate the features from the symbol tree of the best individuals for each run of each learning algorithm. Each symbol tree will have 10 trees of 1 or more symbols as shown in Figure 4.15, this is done 5 times for each of 5 learning algorithms. Therefore, this enumeration will result in at least 250 attributes. Appendix C contains the complete list of the best individuals' symbol trees.

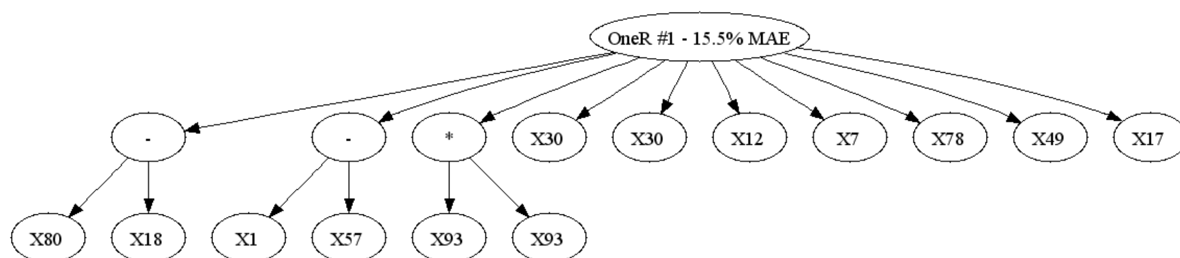

Figure 4.15: An example genetically-constructed condensed symbol tree.

By expanding the condensed view of the symbol tree a more thorough view of the features which were constructed becomes available. Figure 4.16 shows the expanded version of the symbol tree wherein all constructed features (*i.e.*, X93, X7, etc) are expanded.

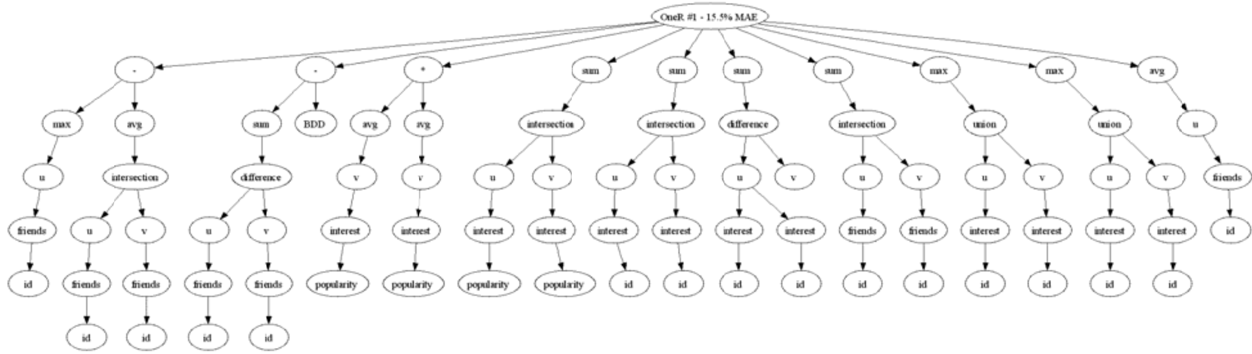

Figure 4.16: An example genetically constructed symbol tree with constructed features expanded.

Table 4.7 presents a partial list of the most commonly constructed features, their code mnemonic, and the feature name. Feature names are descriptively titled. For example, `firstFriendsAgeSum` describes the feature constructed by taking the sum of the ages of u's friends, and `intersectionFriendsIdCount` describes the feature constructed by finding the mean of the IDs in the intersection of u and v's friends where u and v are the two candidate friends.

Table 4.6: Top 10 features ordered by their prevalence in individuals' symbol trees

Feature Name	Feature Mnemonic	Number of Appearances
`firstFriendsAgeSum`	X82	17
`intersectionFriendsIdAvg`	X18	15
`intersectionFriendsIdCount`	X21	15
`intersectionFriendsAgeCount`	X26	11
`intersectionFriendsAgeAvg`	X23	9
`intersectionInterestsIdSum`	X7	7
`unionFriendsAgeSum`	X42	6
`unionFriendsAgeAvg`	X43	6
`firstFriendsIdAvg`	X78	6
`forwardDeletedDistance`	X0	5

Including `forwardDeletedDistance`, there are 7 constructed features that appear 5 times, 8 features that appear 4 times, 14 features that appear 3 times, 17 features that appear twice, 25 features that appear only once, and 28 features that do not appear at all.

4.4 Meta Learning Results

Chapter 3 describes meta-learning as learning from the learning process. This section presents results similar in style to Section 4.1, where the dataset contains 2000 instances which are distributed with 50% positive and 50% negative examples. First, three meta-learning algorithms (AdaBoost, Bagging, and RandomForest) are trained on the dataset and tested on the hold out dataset with 1.5%-positive example distribution. Table 4.7 shows the results of these meta-learning algorithms trained with the base features only.

Table 4.7: Results for meta-learning algorithms trained on base features, tested on hold out data with original distribution

Learning algorithm	AUC	Confusion Matrix		Accuracy	Precision	Recall
AdaBoost	97.6	28	1	92.8	16.4/99.9	96.6/92.7
		143	1828			
Bagging	95.7	28	1	92.95	16.7/99.9	96.6/92.9
		140	1831			
RandomForest	97.8	28	1	91.7	14.5/99.9	96.6/91.6
		165	1806			

As a final test of performance, an ROC curve is generated in order to gauge the overall performance of the learning algorithms. Figure 4.17 shows the ROCs for the various learning algorithms.

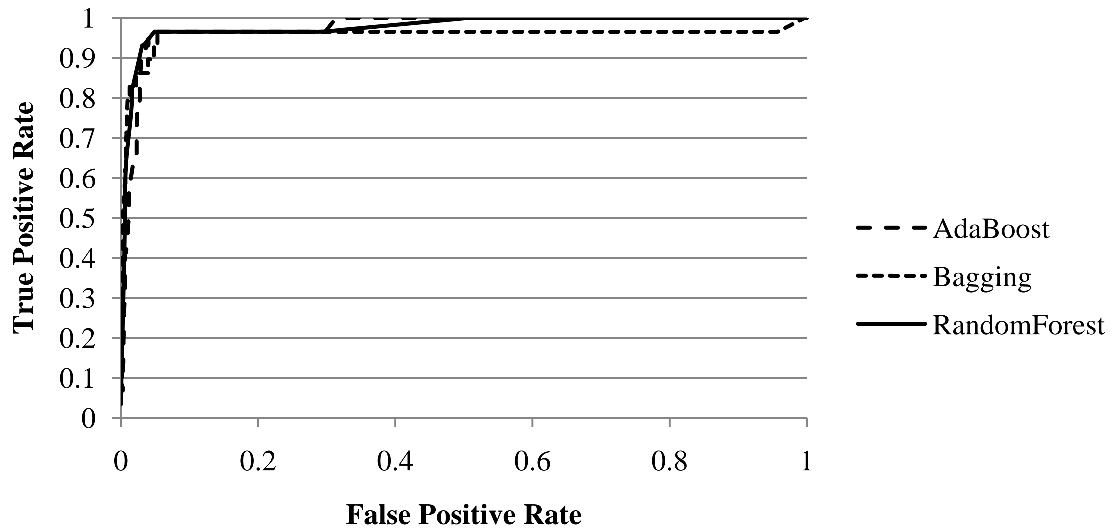

Figure 4.17: ROC curves for learning algorithms trained and cross validated on feature data set.

The area under the curve (AUC) is often used to gauge performance, where the higher the AUC the better the performance. An AUC of 1 is perfect whereas an AUC of 50% is random guess. Table 4.7 shows the AUC values for the ROCs in Figure 4.17. By this metric, the RandomForest meta-learning algorithm performed the best.

CHAPTER 5 - Conclusions and Future Work

In this thesis, I considered the problem of discovering links in a large, incomplete graph. This thesis presented an approach to link prediction that is based on the combination of graph feature analysis and intrinsic attributes of entities. Therefore, the principal claims of this thesis were:

1. By crawling the social network service *LiveJournal* an appropriately large graph can be realized and learnable features can be ascertained.
2. Feature analysis can be achieved on a very large graph by the efficient management of the underlying data structure.
3. Operators within the genetic programming approach can be used to construct new features from primitive features that provide a more learnable description of the graph.
4. Features constructed from a genetic program will improve the performance of learning algorithms for link mining.

By learning from the material in Chapters 1 and 2 and following the methodologies prescribed in Chapter 3 the results presented in Chapter 4 lead to several conclusions. This chapter presents an interpretation of the results and reviews the principal claims of this work in consecutive sections before discussing future directions of this work.

5.1 Large Graph Crawling and Feature Extraction

The structure and results of a large web crawl of a social networks graph was presented in Section 3.1. The aptly named *LJCrawler* crawled 39.024 users, scheduled 770,595 total users, and discovered 2,151,090 interests and 2,992,607 relationships. All of this was accomplished within a time span of about 2½ hours, which equates to 4.09 users per second and 313.69 relations per second. The total uncompressed size of the database is 383.6 MB.

By crawling the social network graph friend of a friend (FOAF) data was retrieved. The FOAF data included the birth-date, location, age, friends, interests, etc., which was efficiently packaged into computable database tuples and stored. Furthermore, from this data 6,000 total candidate pairs were generated (2000 for training, 2000 for testing, and 2000 for validation); these pairs included 108 potential features each giving a total of 648,000 feature computations. These computations were further computed by the genetic program by the methods described in Section 3.3.

Because the pairs were selected at random the entire database was considered by the candidate pair generator. Furthermore, paths between candidate pairs frequently contain unqueried vertices therefore the feature constructor needed to be able to efficiently retrieve and process several data pieces.

Although timing and benchmarking data was not kept for the pair generation and feature construction phases, the 648,000 feature computations did not consume too much wall time (approximately 10 minutes for each set of 2000 pairs). Section 5.2 discusses specific graph feature computation results.

Having demonstrated the ability to quickly and efficiently retrieve vast amounts of social network user information, as well as the ability to ascertain relevant features, I conclude that the necessary requirements have been met to satisfy principal claim 1.

5.2 Efficient Feature Computation on Large Graphs

The efficient computation of features is extremely important because evolutionary algorithms operate by spawning and growing a set of hundreds of unique individuals, and each individual had at least 10 constructed features. Section 3.5 demonstrated that the running time of many of these operations was in $\theta(n^2)$ where n was the size of a user's list of friends or interests, which mostly resulted in an insignificant computation cost. However, the running time of the forward deleted distance and backward deleted distance features were bounded by the size of the graph. The magnitude of the graph made traditional search algorithms, such as Dijkstra's Shortest Path algorithm impractical by incurring an unbearably long average case execution time. This problem was largely relieved with the inclusion of the shortest path approximation algorithm described in Subsection 3.5.1.

Although the theoretical running times have not been annotated, early empirical results show that for a graph size of 50,000 the mean time to find the shortest path between random vertices is 5.44 milliseconds with the approach described in this section compared to a mean time of 229.88 milliseconds for Dijkstra's algorithm. Although this is an approximation the algorithm generally achieves 100% accuracy.

When applied to the feature computation task within this work the shortest path approximation again performed well. Although timing and benchmarking data was not kept for the pair generation and feature construction phases, the 432,000 feature computations did not consume too much wall clock time (approximately 10 minutes for each set of 2000 pairs).

The proportion of computation time for the forward and backward deleted distances relative to the other 108 features is not known. However, extrapolating the result of Dijkstra's algorithm on the 50,000-vertex test graph to the full 770,595-vertex graph shows yields a 3.5 second average execution time. If all other computations were negligible, the execution time for

6000 candidate pairs' forward and backward deleted distances would be approximately 11 hours and 40 minutes ($3.5 \times 2 \times 6000 = 42{,}000$ seconds $\sim 11.\overline{6}$ hours).

Having demonstrated the ability to quickly and efficiently compute critical graph features from a large graph through use of an approximate shortest path algorithm, I conclude that the necessary requirements have been met to satisfy principal claim 2.

5.3 Analysis of Genetically-Constructed Features

This work was mainly involved with the ability of genetic programs to construct new features from primitive features. These newly constructed features were then used to build a classifier. The ultimate goal is to gain insight into the domain by examining these features and their relative performance.

Subsection 4.3.7 illustrates that the genetic programming approach constructed a vast amount of new features. Moreover, the multiple tree symbolic regression approach extended the feature construction approach by incorporating the use of mathematical operators in a symbol tree. I originally expected the features that were to be generated to be very large and complex. For example, I expected a most of the constructed features to contain several layers of set operations on a variety of attributes. However, I was mildly surprised to find out that most of the genetically constructed features were simple set operations of common features. I was further surprised to find that the graph features which were very prevalent in earlier experiments [Hsu06] [Hsu07] [Hsu08] were less prevalent through these experiments. Specifically, Table 4.7, from Chapter 4, shows that the most commonly constructed feature among best individuals was the sum of u's friends' ages, whereas the first graph feature, forward deleted distance, was 10th on the list.

Overall some interesting patterns emerged from these results. First, the intersection set operation played a large role in the fitness of individuals, while the union set operation played a minor role and the difference set operator was not identified in the top 10. Intuitively, this shows that what friends have in common is more telling than what they do not have in common. Second, the use of the friends attribute exists in all but 2 of the top 10 constructed features. This evidence, coupled with the prevalence of the intersection set operator, shows that common

friends are a good indicator of a relationship. Third, the sub-attributes `age` and `id` are the only two sub-attributes identified for the friendship attribute and because they occur equally their relative prevalence is moot. Finally, only a single feature containing interest data made the top 10; this fact falls in line with earlier observations that show interest data alone is a relatively poor predictor of friendship.

Having demonstrated the ability for a genetic program to construct features from a large and complex graph-dataset, I conclude that the necessary requirements have been met to satisfy principal claim 3.

5.4 Prediction Performance of Genetically-Constructed Features

The final claim of this thesis is that if the genetically-constructed features are used to train classifiers from various learning algorithms then the performance of these classifiers will be better than the classifiers training on base features alone. Chapter 4 presented the results of several classifiers. First, classifiers were trained using the base features only. Second, features were constructed from a genetic program using a single symbol tree, and these features were used to train and test a classifier. Third, features were constructed from a genetic program using multiple symbol trees, and these features were similarly used to train and test a classifier. Finally, meta-learning methods were trained and tested on the base features.

The classifiers trained using base features only exhibited performance similar to that in previous work [Hsu06] [Hsu07]. These results are used as the baseline prediction performance.

Unlike in the genetic programs with multiple regression trees, the single regression tree method did not converge. Rather, the results maintained a steady error rate. I believe this is because the single symbol tree equally modified the distribution of all of the learning algorithm's inputs thereby making the symbol trees modifications meaningless. For example, if two training tuples t_1 and t_2 each contained data $t_1 = \langle 2,3,5, yes \rangle$, and $t_2 = \langle 10,13,11, no \rangle$ then a simple classifier would say all future tuples with values greater than 7 should be labeled no otherwise they should be labeled yes. To complete this example, consider a single symbol tree `(+ x x)` which transforms t_1 into $\langle 4,6,10, yes \rangle$ and $\langle 20,26,22, yes \rangle$. Again, a simple classifier would

label all future tuples with values greater than $(7 + 7 =)$ 14 to be no, otherwise, they would be labeled yes.

Feature construction from multiple symbol trees alleviates this problem because with multiple symbol trees the data is not all altered in exactly the same way. This causes a clear discrimination among classes to emerge and prediction performance to increase.

Figure 5.1 shows a comparison between the performance of traditional learning algorithms with and without genetically constructed features and the newer meta-learning algorithms, which were training on base features. The leftmost 5 columns represent the traditional learning algorithms, and the rightmost 3 columns represent the meta learning algorithms. The genetically-constructed features results shown here are the average, best and final individuals from earlier experiments (*i.e.*, from Tables 4.6 and 4.8).

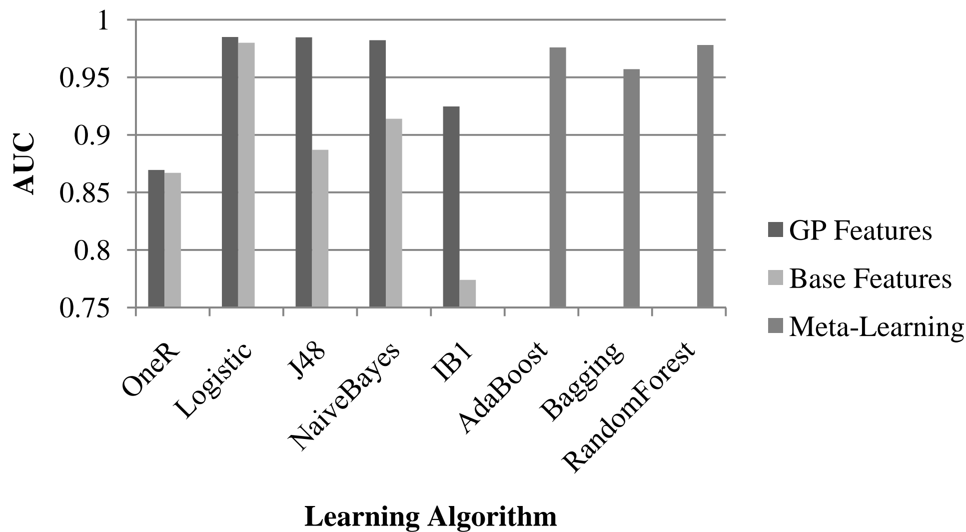

Figure 5.1: Comparison of results from classifiers trained by (1) genetically-constructed features (2) base features and (3) meta-learning algorithms.

The final performance of classifiers trained by genetically-constructed features outperforms classifiers trained by base features in every instance, and the top 3 genetic results outperform even the best meta-learning algorithm tested. Therefore, I conclude that the necessary requirements have been met to satisfy principal claim 4.

5.5 Future Work

Link mining is situated at the intersection of graph theory, machine learning, and web mining. This research is potentially useful in a wide range of application areas including bio-informatics, bibliographic analysis, financial analysis, national security, social network analysis, and internet search to name a few. While my research is focused more on the theoretical aspects of this topic than in the applicative possibilities, a positive outcome of my work is that it has already been adapted to the bioinformatics domain to study the interactions of proteins. [Par07] Other promising research includes ontology engineering for interest intersections [Bah08], and market basket analysis for interest and community membership [Alj08].

In the immediate future the approximate shortest path algorithm should be finalized and theoretically examined. I believe that this approach to searching through graphs has implications in many fields. Similar work on link mining on the web [Wen08] [Wen08a] has yielded promising results, and previous work in genetic and evolutionary computation [Hsu06a] [Hsu07a] provides some direction for future work.

Finally, one limitation of this work was the expressiveness of the constructed features. One avenue for future research is to allow the genetic program more degrees of freedom in its construction of features. This could be done by adjusting the evolution parameters in system, or by using alternative fitness measures.

A final criticism of this work originates from a potential bias implicit in the network methodology. One might argue that the removal of a friendship link to compute alternate paths and then the immediate re-addition of that link creates a favorable bias. With that in mind, alternative studies from Taskar, Getoor and Koller remove all the links they wish to predict without any re-addition [Tas02], [Tas01], [Get01]. I understand the potential bias inherent in this approach, but I decided to expand on my previous published work for continuity-sake.

The research field of link mining is still in its infancy, and there are undoubtedly many exciting breakthroughs to be made in the search, retrieval, annotation and explanation of relationships in all types of datasets.

Bibliography

[Alj08] W. Aljandal, V. Bahirwani, D. Caragea, W. H. Hsu, and T. Weninger, "Validation-based normalization and selection of interestingness measures for association rules," in *Artificial Neural Networks In Engineering*, St. Louis, MO, 2008.

[Bah08] V. Bahirwani, D. Caragea, W. Aljandal, and W. H. Hsu, "Ontology Engineering and Feature Construction for Predicting Friendship Links in the Live Journal Social Network," in *Proceedings of the KDD 2008 Second Workshop on Social Network Mining and Analysis*, Las Vegas, NV, 2008.

[Bel61] R. Bellman, *Adaptive control processes: A guided tour*. Princeton, NJ: Princeton University Press, 1961.

[Ben96] H. Bensusan and I. Kuscu, "Constructive Induction using Genetic Programming," in *International Conference on Machine Learning (ICML) Evolutionary computing and Machine Learning Workshop*, 1996.

[Blu87] A. Blumer, J. Blumer, D. Haussler, R. Mcconnell, and A. Ehrenfeucht, "Complete inverted files for efficient text retrieval and analysis," vol. 34, pp. 578-595, 1987.

[Blu97] A. L. Blum and P. Langley, "Selection of Relevant Features and Examples in Machine Learning," *Artificial Intelligence*, pp. 245-271, 1997.

[Bre01] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5-32, 2001.

[Bre02] L. Breiman and A. Cutler. (2002) Random Forests. [Online]. http://stat-www.berkeley.edu/users/breiman/RandomForests/cc_home.htm

[Bre84] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*. Belmont, CA: Wadsworth International Group, 1984.

[Bre96] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123-140, 1996.

[Bri98] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107-117, Apr. 1998.

[Car93] C. Cardie, "Using Decision Trees to Improve Case-Based Learning," in *International Conference on Machine Learning (ICML)*, 1993, pp. 25-32.

[Car94] R. Caruna and D. Freitag, "How Useful is Relevance?," in *AAAI Fall Symposium on Relevance*, New Orleans, 1994.

[Cha08] J. C.-W. Chan and D. Paelinckx, "Evaluation of Random Forest and Adaboost tree-based ensemble classification and spectral band selection for ecotope mapping using airborne hyperspectral imagery," *Remote Sensing of Environment*, vol. 112, no. 6, pp. 2999-2011, Jun. 2008.

[Cha85] G. Chartrand, *Introductory Graph Theory*. New York: Dover, 1985.

[Col92] W. J. Collins, *Data Structures: An Object-Oriented Approach*. Addison-Wesley, 1992.

[Dam08] M. Damo, "The Social Net of Digg: Structural Analysis and Prediction," Milan Polytechnic Thesis, 2008.

[Dan01] D. Dand, H. Mannila, and P. Smyth, *Principles of Data Mining*. MIT Press, 2001.

[Das00] M. Dash and H. Liu, "Feature Selection for Clustering," in *Pacific Asia Conference on Knowledge Discovery in Data Mining (PAKDD)*, 2000, pp. 110-121.

[DiB99] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[DyJ00] J. G. Dy and C. E. Brodley, "Feature Subset Selection and Order Identification for Unsupervised Leaderning," in *International Conference on Machine Learning (ICML)*, 2000, pp. 247-254.

[Fis87] D. H. Fisher, "Knowledge acquisition via incremental conceptual clustering," *Machine Learning*, vol. 2, no. 2, pp. 139-172, Sep. 1987.

[Fre96] Y. Freund and R. Schapire, "Experiments with a new boosting algorithm," in *International Conference on Machine Learning (ICML)*, 1996, pp. 148-156.

[Gam98] J. Gama and P. Brazdil, "Constructive Induction on Continuous Spaces," in *Feature Extraction, Construction and Selection: A Data Mining Perspective*, H. Motoda and H. Liu, Eds. Boston: Kluwer Academic Publishers, 1998, ch. 18, pp. 289-303.

[Get01] L. Getoor, N. Friedman, D. Koller, and B. Taskar, "Learning Probabilistic Models of Relational Structure," in *International Conference on Machine Learning*, 2001.

[Get03] L. Getoor, "Link Mining: A New Data Mining Challenge," *SIGKDD Explorations*, vol. 4, no. 2, pp. 1-6, 2003.

[Gol89] D. E. Goldberg, *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, 1989.

[Gos05] O. Gospodnetic and E. Hatcher, *Lucene in Action*. Greenwich, CT: Manning, 2005.

[Han06] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, 2nd ed., J. Gray, Ed. San Francisco, CA: Morgan Kauffman, 2006.

[Har02] D. Harel and Y. Koren, "Graph drawing by high-dimensional embedding," *LNCS*, pp. 207-219, 2002.

[Hsu06] W. H. Hsu, A. King, M. S. R. Paradesi, T. Pydimarri, and T. Weninger, "Collaborative and Structural Recommendation of Friends using Weblog-based Social Network Analysis," in *AAAI Spring Symposium on Computatational Approaches to Analyzing Weblogs (CAAW)*, 2006.

[Hsu06a] W. H. Hsu, A. L. King, M. S. R. Paradesi, T. Pydimarri, and T. Weninger, "Evolutionary Data Mining For Link Analysis: Preliminary Experiments On A Social Network Test Bed," in *Genetic and Evolutionary Computation Conference*, Seattle, WA, 2006.

[Hsu07] W. H. Hsu, J. Lancaster, M. S. R. Paradesi, and T. Weninger, "Structural Link Analysis from User Profiles and Friends Networks: A Feature Construction Approach," in *International Conference on Weblogs and Social Media (ICWSM)*, Boulder, CO, 2007, pp. 75-80.

[Hsu07a] W. H. Hsu, J. Lancaster, M. S. R. Paradesi, and T. Weninger, "Collaborative and Structural Recommendation of Friends using Weblog-Based Social Network Analysis," in *Genetic and Evolutionary Computation Conference*, London, UK, 2007.

[Hsu08] W. H. Hsu, T. Weninger, and M. S. R. Paradesi, "Predicting links and link change in friends networks: Supervised time series learning with imbalanced data," in *Artificial Neural Networks in Engineering*, St. Louis, MO, 2008.

[Kau01] "Drawing Graphs: Methods and Models," *LNCS*, vol. 2025, 2001.

[Kim00] Y. Kim, W. Street, and F. Menczer, "Feature Selection for Unzupervised Learning via Evolutionary Search," in *International Conference on Knowledge Discovery and Data Mining (ACM SIGKDD)*, 2000, pp. 365-369.

[Kir83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671-680, 1983.

[Kir92] K. Kira and L. Rendell, "The feature selection problem: Traditional methods and a new algorithm," in *National Conference on Artificial Intelligence*, 1992, pp. 129-134.

[Kol01] D. Koller, "Representation, Reasoning and Learning," *IJCAI Computers and Thought Award Lecture*, 2001.

[Koz92] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[Koz98] J. R. Koza, *Genetic Programming II*. Cambridge, MA: MIT Press, 1998.

[Kra02] K. Krawiec, "Genetic Programming-based Construction of Features for Machine Learning and Knowledge Discovery Tasks," *Genetic Programming and Evolvable Machines*, vol. 3, pp. 329-343, 2002.

[Kre02] V. E. Krebs, "Mapping Networks of Terrorist Cells," *Connections*, 2002.

[Kub97] M. Kubat, R. Holte, and S. Matwin, "Learning when negative examples abound," in *ECML, Lecture Notes in Artificial Intelligence*. Springer Verlag, 1997.

[Lan94] P. Langley, "Selection of relevant features in machine learning," in *AAAI Fall symposium on relevance*, 1994, pp. 140-144.

[Liu96] H. Liu and R. Setiono, "Feature Selection and Classification - A Probabilistic Wrapper Approach," in *International Conference on Industrial and Engineering Applciations*, 1996, pp. 419-424.

[Liu98] H. Liu and H. Motoda, Eds., *Feature Extraction, Construction and Selection: A Data Mining Perspective*. Boston: Kluwer Academic Publishing, 1998.

[Luk01] S. Luke. (2001) ECJ 7: An EC and GP system in Java. [Online]. http://www.cs.umd.edu/projects/plus/ec/ecj/

[Mat89] C. J. Matheus and L. A. Rendell, "Constructive Induction On Decision Trees," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 1989, pp. 645-650.

[Mer95] C. J. Merz, "Dynamical Selection of Learning Algorithms," in *Learning from Data: Artificial Intelligence and Statistics*, D. Fisher and H. J. Lenz, Eds. Springer-Verlag, 1995.

[Mic83] R. S. Michalski, *A Theory and Methodology of Inductive Learning*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kaufmann, 1983.

[Mit80] T. M. Mitchell, "The Need for Biases in Learning Generalizations," Technical Report CBM-TR-117, 1980.

[Mit82] T. M. Mitchell, "Generalization as Search," *Artificial Intelligence*, vol. 18, no. 2, 1982.

[Mit83] T. Mitchell, "Learning and Problem-Solving," in *International Joint Conference on Artificial Intelligence (IJCAI)*, Karlsruhe, Germany, 1983.

[Mit97] T. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[Mor93] M. Morris, "Epidemiology and Social Networks," *Sociological Methods and Research*, vol. 22, no. 1, pp. 99-126, 1993.

[Mot02] H. Motodo and H. Liu, "Feature Selection, Extraction and Construction," in *Pacific-Asia Conference for Advances in Knowledge Discovery and Data Mining*, Taipei, Taiwan, 2002.

[Ols99] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley DB," in *USENIX Annual Technical Conference*, Monterey, CA, 1999, pp. 183-194.

[Pag90] G. Pagallo and D. Haussler, "Boolean Feature Discovery in Empirical Learning," *Machine Learning*, vol. 5, no. 1, pp. 71-99, Mar. 1990.

[Par07] M. S. R. Paradesi, D. Caragea, and W. H. Hsu, "Structural Prediction of Protein-Protein Interactions in Saccharomyces cerevisiae," in *IEEE 7th International Symposium on BioInformatics and BioEngineering*, Boston, MA, 2007, pp. 1270-1274.

[Per03] B. B. Perry, "A Genetic Algorithm for Learning Bayesian Network Adjacency Matrices from Data," Kansas State University Thesis, 2003.

[Pre98] B. Preiss, *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, 1998.

[Pro05] F. Provost and J. Langford. (2005, Feb.) Machine Learning (Theory). [Online]. http://hunch.net/?p=21

[Qui86] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, pp. 81-106, 1986.

[Qui92] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann, 1992.

[Rao95] R. B. Rao, D. Gordon, and W. Spears, "For every generalization action, is there really an equal and opposite reaction? Analysis of the conservation law for generalization performance," in *International Conference on Machine Learning*, 1995, pp. 471-479.

[Ren88] L. Rendell, "Learning Hard Concepts," in *European Working Session on Learning*, Turing Institute, Glasgow, 1988.

[Ren90] L. A. Rendell and R. Seshu, "Learning hard concepts through constructive induction: framework," *Computational Intelligence*, vol. 6, no. 4, pp. 247-270, 1990.

[Ren90a] L. A. Rendell and H. Cho, "Empirical Learning as a Function of Concept Character," *Machine Learning*, vol. 5, pp. 267-298, 1990.

[Rus03] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Upper Saddle River, NJ: Pearson Education, Inc., 2003.

[Sam59] A. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *Computation and Intelligence*, 1959.

[Sch02] R. E. Schapire, "The Boosting Approach to Machine Learning: An Overview," in *Mathematical Sciences Research Institute (MSRI) Workshop on Nonlinear Estimation and Classification*, 2002.

[Sch94] C. Schaffer, "A Conservation Law for Generalization Performance," in *International Conference on Machine Learning (ICML)*, 1994, pp. 259-265.

[Set01] R. Setiono and H. Liu, "Feature Extraction via Neural Networks," in *Feature Extraction Construction and Selection: A Data Mining Perspective*. Kluwer Academic

Publishing, 2001.

[Tal99] L. Talavera, "Feature Selection as a Preprocessing Step for Hierarchical Clustering," in *International Conference on Machine Learning (ICML)*, 1999, pp. 389-397.

[Tal99a] L. Talavera, "Feature Selection as Retrospective Pruning in Hierarchical Clustering," in *Symposium on Intelligent Data Analysis (IDA)*, 1999, pp. 75-86.

[Tan06] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Addison-Wesley, 2006.

[Tas01] B. Taskar, E. Segal, and D. Koller, "Probabilistic Classification and Clustering in Relational Data," in *International Joint Conference on Artificial Intelligence*, Seattle, WA, 2001, pp. 870-876.

[Tas02] B. Taskar, P. Abbeel, and D. Koller, "Discriminative probabilistic models for relational data," in *Uncertainty in Artificial Intelligence*, 2002.

[Thr98] S. Thrun, "Lifelong Learning Algorithms," *Learning to Learn*, vol. 8, pp. 181-209, 1998.

[Vil02] R. Vilalta and Y. Drissi, "A Perspective View and Survey of Meta-Learning," *Artificial Intelligence Review*, vol. 18, no. 2, pp. 77-95, 2002.

[Was05] S. Wasserman, J. Scott, and P. J. Carrinton, "Introduction," in *Models and Methods in Social Network Analysis*, M. Granovetter, Ed. Cambridge, NY: Cambridge University Press, 2005, ch. 1, pp. 1-5.

[Was07] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*, M. Granovetter, Ed. Cambridge, NY: Cambridge University Press, 2007.

[Wat69] S. Watanabe, *Knowing and Guessing: A Quantitative Study of Inference and Information*. New York: Wiley, 1969.

[Wei95] S. M. Weiss and N. Indurkhya, "Rule-based Machine Learning Methods for Functional Prediction," *Journal of Artificial Intelligence Research*, vol. 3, pp. 383-403, 1995.

[Wen08] T. Weninger and W. H. Hsu, "Web Content Extraction Through Histogram Clustering," in *Artificial Neural Networks in Engineering*, St. Louis, MO, 2008.

[Wen08a] T. Weninger and W. H. Hsu, "Text Extraction from the Web via Text-To-Tag Ratio," in *Database and Expert Systems Applications Workshop on Text-based Information Retrieval*, Turin, Italy, 2008.

[Wik08] Wikipedia. (2008, Nov.) Wikipedia. [Online]. http://en.wikipedia.org/wiki/Receiver_operating_characteristic

[Wik08a] Wikipedia. (2008) Occom's Razor. [Online]. en.wikipedia.org/wiki/Occam's_razor

[Wit99] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. San Francisco, CA: Morgan Kaufmann, 1999.

[Wol92] D. H. Wolpert, "Stacked Generalization," *Neural Networks*, vol. 5, pp. 241-259, 1992.

[Wys80] N. Wyse, R. Dubes, and A. K. Jain, "A critical evaluation of intrinsic dimensionality algorithms," in *Pattern Recognition in Practice*, E. S. Gelsema and L. N. Kanal, Eds. Morgan Kaufmann, 1980, pp. 415-425.

[Zhe98] Z. Zheng, "A Comparision of Constructing Different Types of New Features for Decision Tree Learning," in *Feature Extraction, Construction and Selection: A Data*

Mining Perspective, H. Liu and H. Motoda, Eds. Boston, MA: Kluwer Academic Publishing, 1998, ch. 15, pp. 239-255.

Appendix A - Complete Feature Set

```
1.   @RELATION    linkmining
2.   @ATTRIBUTE   fdd                                  NUMERIC
3.   @ATTRIBUTE   bdd                                  NUMERIC
4.   @ATTRIBUTE   indegreeu                            NUMERIC
5.   @ATTRIBUTE   outdegreeu                           NUMERIC
6.   @ATTRIBUTE   indegreev                            NUMERIC
7.   @ATTRIBUTE   outdegreev                           NUMERIC
8.   @ATTRIBUTE   euclidDist                           NUMERIC
9.   @ATTRIBUTE   intersectioninterestsidsum           NUMERIC
10.  @ATTRIBUTE   intersectioninterestsidavg           NUMERIC
11.  @ATTRIBUTE   intersectioninterestsidmin           NUMERIC
12.  @ATTRIBUTE   intersectioninterestsidmax           NUMERIC
13.  @ATTRIBUTE   intersectioninterestsidcount         NUMERIC
14.  @ATTRIBUTE   intersectioninterestspopularitysum   NUMERIC
15.  @ATTRIBUTE   intersectioninterestspopularityavg   NUMERIC
16.  @ATTRIBUTE   intersectioninterestspopularitymin   NUMERIC
17.  @ATTRIBUTE   intersectioninterestspopularitymax   NUMERIC
18.  @ATTRIBUTE   intersectioninterestspopularitycount NUMERIC
19.  @ATTRIBUTE   intersectionfriendsidsum             NUMERIC
20.  @ATTRIBUTE   intersectionfriendsidavg             NUMERIC
21.  @ATTRIBUTE   intersectionfriendsidmin             NUMERIC
22.  @ATTRIBUTE   intersectionfriendsidmax             NUMERIC
23.  @ATTRIBUTE   intersectionfriendsidcount           NUMERIC
24.  @ATTRIBUTE   intersectionfriendsagesum            NUMERIC
25.  @ATTRIBUTE   intersectionfriendsageavg            NUMERIC
26.  @ATTRIBUTE   intersectionfriendsagemin            NUMERIC
27.  @ATTRIBUTE   intersectionfriendsagemax            NUMERIC
28.  @ATTRIBUTE   intersectionfriendsagecount          NUMERIC
29.  @ATTRIBUTE   unioninterestsidsum                  NUMERIC
30.  @ATTRIBUTE   unioninterestsidavg                  NUMERIC
31.  @ATTRIBUTE   unioninterestsidmin                  NUMERIC
32.  @ATTRIBUTE   unioninterestsidmax                  NUMERIC
33.  @ATTRIBUTE   unioninterestsidcount                NUMERIC
34.  @ATTRIBUTE   unioninterestspopularitysum          NUMERIC
35.  @ATTRIBUTE   unioninterestspopularityavg          NUMERIC
36.  @ATTRIBUTE   unioninterestspopularitymin          NUMERIC
37.  @ATTRIBUTE   unioninterestspopularitymax          NUMERIC
38.  @ATTRIBUTE   unioninterestspopularitycount        NUMERIC
39.  @ATTRIBUTE   unionfriendsidsum                    NUMERIC
40.  @ATTRIBUTE   unionfriendsidavg                    NUMERIC
41.  @ATTRIBUTE   unionfriendsidmin                    NUMERIC
42.  @ATTRIBUTE   unionfriendsidmax                    NUMERIC
43.  @ATTRIBUTE   unionfriendsidcount                  NUMERIC
44.  @ATTRIBUTE   unionfriendsagesum                   NUMERIC
45.  @ATTRIBUTE   unionfriendsageavg                   NUMERIC
46.  @ATTRIBUTE   unionfriendsagemin                   NUMERIC
47.  @ATTRIBUTE   unionfriendsagemax                   NUMERIC
48.  @ATTRIBUTE   unionfriendsagecount                 NUMERIC
49.  @ATTRIBUTE   differenceinterestsidsum             NUMERIC
50.  @ATTRIBUTE   differenceinterestsidavg             NUMERIC
51.  @ATTRIBUTE   differenceinterestsidmin             NUMERIC
52.  @ATTRIBUTE   differenceinterestsidmax             NUMERIC
53.  @ATTRIBUTE   differenceinterestsidcount           NUMERIC
```

```
54.     @ATTRIBUTE  differenceinterestspopularitysum       NUMERIC
55.     @ATTRIBUTE  differenceinterestspopularityavg       NUMERIC
56.     @ATTRIBUTE  differenceinterestspopularitymin       NUMERIC
57.     @ATTRIBUTE  differenceinterestspopularitymax       NUMERIC
58.     @ATTRIBUTE  differenceinterestspopularitycount     NUMERIC
59.     @ATTRIBUTE  differencefriendsidsum                 NUMERIC
60.     @ATTRIBUTE  differencefriendsidavg                 NUMERIC
61.     @ATTRIBUTE  differencefriendsidmin                 NUMERIC
62.     @ATTRIBUTE  differencefriendsidmax                 NUMERIC
63.     @ATTRIBUTE  differencefriendsidcount               NUMERIC
64.     @ATTRIBUTE  differencefriendsagesum                NUMERIC
65.     @ATTRIBUTE  differencefriendsageavg                NUMERIC
66.     @ATTRIBUTE  differencefriendsagemin                NUMERIC
67.     @ATTRIBUTE  differencefriendsagemax                NUMERIC
68.     @ATTRIBUTE  differencefriendsagecount              NUMERIC
69.     @ATTRIBUTE  firstinterestsidsum                    NUMERIC
70.     @ATTRIBUTE  firstinterestsidavg                    NUMERIC
71.     @ATTRIBUTE  firstinterestsidmin                    NUMERIC
72.     @ATTRIBUTE  firstinterestsidmax                    NUMERIC
73.     @ATTRIBUTE  firstinterestsidcount                  NUMERIC
74.     @ATTRIBUTE  firstinterestspopularitysum            NUMERIC
75.     @ATTRIBUTE  firstinterestspopularityavg            NUMERIC
76.     @ATTRIBUTE  firstinterestspopularitymin            NUMERIC
77.     @ATTRIBUTE  firstinterestspopularitymax            NUMERIC
78.     @ATTRIBUTE  firstinterestspopularitycount          NUMERIC
79.     @ATTRIBUTE  firstfriendsidsum                      NUMERIC
80.     @ATTRIBUTE  firstfriendsidavg                      NUMERIC
81.     @ATTRIBUTE  firstfriendsidmin                      NUMERIC
82.     @ATTRIBUTE  firstfriendsidmax                      NUMERIC
83.     @ATTRIBUTE  firstfriendsidcount                    NUMERIC
84.     @ATTRIBUTE  firstfriendsagesum                     NUMERIC
85.     @ATTRIBUTE  firstfriendsageavg                     NUMERIC
86.     @ATTRIBUTE  firstfriendsagemin                     NUMERIC
87.     @ATTRIBUTE  firstfriendsagemax                     NUMERIC
88.     @ATTRIBUTE  firstfriendsagecount                   NUMERIC
89.     @ATTRIBUTE  secondinterestsidsum                   NUMERIC
90.     @ATTRIBUTE  secondinterestsidavg                   NUMERIC
91.     @ATTRIBUTE  secondinterestsidmin                   NUMERIC
92.     @ATTRIBUTE  secondinterestsidmax                   NUMERIC
93.     @ATTRIBUTE  secondinterestsidcount                 NUMERIC
94.     @ATTRIBUTE  secondinterestspopularitysum           NUMERIC
95.     @ATTRIBUTE  secondinterestspopularityavg           NUMERIC
96.     @ATTRIBUTE  secondinterestspopularitymin           NUMERIC
97.     @ATTRIBUTE  secondinterestspopularitymax           NUMERIC
98.     @ATTRIBUTE  secondinterestspopularitycount         NUMERIC
99.     @ATTRIBUTE  secondfriendsidsum                     NUMERIC
100.    @ATTRIBUTE  secondfriendsidavg                     NUMERIC
101.    @ATTRIBUTE  secondfriendsidmin                     NUMERIC
102.    @ATTRIBUTE  secondfriendsidmax                     NUMERIC
103.    @ATTRIBUTE  secondfriendsidcount                   NUMERIC
104.    @ATTRIBUTE  secondfriendsagesum                    NUMERIC
105.    @ATTRIBUTE  secondfriendsageavg                    NUMERIC
106.    @ATTRIBUTE  secondfriendsagemin                    NUMERIC
107.    @ATTRIBUTE  secondfriendsagemax                    NUMERIC
108.    @ATTRIBUTE  secondfriendsagecount                  NUMERIC
109.    @ATTRIBUTE  friend                                 {yes,no}
```

Appendix B - ECJ Parameters File

```
parent.0 = ../../../gp/koza/koza.params

pop.subpop.0.species.ind.numtrees = 10

# "The result-producing branch" (the "third" tree in Koza-I p. 538)
pop.subpop.0.species.ind.tree.0 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.0.tc = tc0
# "ADF0 body" (the "first" tree in Koza-I p. 538)
pop.subpop.0.species.ind.tree.1 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.1.tc = tc1
# "ADF1 body" (the "second" tree in Koza-I p. 538)
pop.subpop.0.species.ind.tree.2 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.2.tc = tc2

pop.subpop.0.species.ind.tree.3 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.3.tc = tc3

pop.subpop.0.species.ind.tree.4 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.4.tc = tc4

pop.subpop.0.species.ind.tree.5 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.5.tc = tc5

pop.subpop.0.species.ind.tree.6 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.6.tc = tc6

pop.subpop.0.species.ind.tree.7 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.7.tc = tc7

pop.subpop.0.species.ind.tree.8 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.8.tc = tc8

pop.subpop.0.species.ind.tree.9 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.9.tc = tc9

pop.subpop.0.species.ind.tree.10 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.10.tc = tc10

# Now, let's define what tc0, tc1, and tc2 are.
# Each has a different function set, f0, f1, and f2

gp.tc.size = 11

gp.tc.0 = ec.gp.GPTreeConstraints
gp.tc.0.name = tc0
gp.tc.0.fset = f0
gp.tc.0.returns = nil
gp.tc.0.init = ec.gp.koza.GrowBuilder
gp.tc.0.init.growp = 0.5
gp.tc.0.init.min = 2
gp.tc.0.init.max = 6
```

```
gp.tc.1 = ec.gp.GPTreeConstraints
gp.tc.1.name = tc1
gp.tc.1.fset = f0
gp.tc.1.returns = nil
gp.tc.1.init = ec.gp.koza.GrowBuilder
gp.tc.1.init.growp = 0.5
gp.tc.1.init.min = 2
gp.tc.1.init.max = 6

gp.tc.2 = ec.gp.GPTreeConstraints
gp.tc.2.name = tc2
gp.tc.2.fset = f0
gp.tc.2.returns = nil
gp.tc.2.init = ec.gp.koza.GrowBuilder
gp.tc.2.init.growp = 0.5
gp.tc.2.init.min = 2
gp.tc.2.init.max = 6

gp.tc.3 = ec.gp.GPTreeConstraints
gp.tc.3.name = tc3
gp.tc.3.fset = f0
gp.tc.3.returns = nil
gp.tc.3.init = ec.gp.koza.GrowBuilder
gp.tc.3.init.growp = 0.5
gp.tc.3.init.min = 2
gp.tc.3.init.max = 6

gp.tc.4 = ec.gp.GPTreeConstraints
gp.tc.4.name = tc4
gp.tc.4.fset = f0
gp.tc.4.returns = nil
gp.tc.4.init = ec.gp.koza.GrowBuilder
gp.tc.4.init.growp = 0.5
gp.tc.4.init.min = 2
gp.tc.4.init.max = 6

gp.tc.5 = ec.gp.GPTreeConstraints
gp.tc.5.name = tc5
gp.tc.5.fset = f0
gp.tc.5.returns = nil
gp.tc.5.init = ec.gp.koza.GrowBuilder
gp.tc.5.init.growp = 0.5
gp.tc.5.init.min = 2
gp.tc.5.init.max = 6

gp.tc.6 = ec.gp.GPTreeConstraints
gp.tc.6.name = tc6
gp.tc.6.fset = f0
gp.tc.6.returns = nil
gp.tc.6.init = ec.gp.koza.GrowBuilder
gp.tc.6.init.growp = 0.5
gp.tc.6.init.min = 2
gp.tc.6.init.max = 6

gp.tc.7 = ec.gp.GPTreeConstraints
gp.tc.7.name = tc7
gp.tc.7.fset = f0
```

```
gp.tc.7.returns = nil
gp.tc.7.init = ec.gp.koza.GrowBuilder
gp.tc.7.init.growp = 0.5
gp.tc.7.init.min = 2
gp.tc.7.init.max = 6

gp.tc.8 = ec.gp.GPTreeConstraints
gp.tc.8.name = tc8
gp.tc.8.fset = f0
gp.tc.8.returns = nil
gp.tc.8.init = ec.gp.koza.GrowBuilder
gp.tc.8.init.growp = 0.5
gp.tc.8.init.min = 2
gp.tc.8.init.max = 6

gp.tc.9 = ec.gp.GPTreeConstraints
gp.tc.9.name = tc9
gp.tc.9.fset = f0
gp.tc.9.returns = nil
gp.tc.9.init = ec.gp.koza.GrowBuilder
gp.tc.9.init.growp = 0.5
gp.tc.9.init.min = 2
gp.tc.9.init.max = 6

gp.tc.10 = ec.gp.GPTreeConstraints
gp.tc.10.name = tc10
gp.tc.10.fset = f0
gp.tc.10.returns = nil
gp.tc.10.init = ec.gp.koza.GrowBuilder
gp.tc.10.init.growp = 0.5
gp.tc.10.init.min = 2
gp.tc.10.init.max = 6

gp.fs.size = 1
gp.fs.0.name = f0
gp.fs.0.info = ec.gp.GPFuncInfo
gp.fs.0.size = 109
gp.fs.0.func.0 = ec.app.linkmining.regression.func.Add
gp.fs.0.func.0.nc = nc2
gp.fs.0.func.1 = ec.app.linkmining.regression.func.Mul
gp.fs.0.func.1.nc = nc2
gp.fs.0.func.2 = ec.app.linkmining.regression.func.Sub
gp.fs.0.func.2.nc = nc2
gp.fs.0.func.3 = ec.app.linkmining.regression.func.Div
gp.fs.0.func.3.nc = nc2

gp.fs.0.func.4 = ec.app.linkmining.regression.func.X0
gp.fs.0.func.4.nc = nc0
gp.fs.0.func.5 = ec.app.linkmining.regression.func.X1
gp.fs.0.func.5.nc = nc0
gp.fs.0.func.6 = ec.app.linkmining.regression.func.X2
gp.fs.0.func.6.nc = nc0
gp.fs.0.func.7 = ec.app.linkmining.regression.func.X3
gp.fs.0.func.7.nc = nc0
gp.fs.0.func.8 = ec.app.linkmining.regression.func.X4
gp.fs.0.func.8.nc = nc0
gp.fs.0.func.9 = ec.app.linkmining.regression.func.X5
```

```
gp.fs.0.func.9.nc = nc0
gp.fs.0.func.10 = ec.app.linkmining.regression.func.X6
gp.fs.0.func.10.nc = nc0
gp.fs.0.func.11 = ec.app.linkmining.regression.func.X7
gp.fs.0.func.11.nc = nc0
gp.fs.0.func.12 = ec.app.linkmining.regression.func.X8
gp.fs.0.func.12.nc = nc0
gp.fs.0.func.13 = ec.app.linkmining.regression.func.X9
gp.fs.0.func.13.nc = nc0
gp.fs.0.func.14 = ec.app.linkmining.regression.func.X10
gp.fs.0.func.14.nc = nc0
gp.fs.0.func.15 = ec.app.linkmining.regression.func.X11
gp.fs.0.func.15.nc = nc0
gp.fs.0.func.16 = ec.app.linkmining.regression.func.X12
gp.fs.0.func.16.nc = nc0
gp.fs.0.func.17 = ec.app.linkmining.regression.func.X13
gp.fs.0.func.17.nc = nc0
gp.fs.0.func.18 = ec.app.linkmining.regression.func.X14
gp.fs.0.func.18.nc = nc0
gp.fs.0.func.19 = ec.app.linkmining.regression.func.X15
gp.fs.0.func.19.nc = nc0
gp.fs.0.func.20 = ec.app.linkmining.regression.func.X16
gp.fs.0.func.20.nc = nc0
gp.fs.0.func.21 = ec.app.linkmining.regression.func.X17
gp.fs.0.func.21.nc = nc0
gp.fs.0.func.22 = ec.app.linkmining.regression.func.X18
gp.fs.0.func.22.nc = nc0
gp.fs.0.func.23 = ec.app.linkmining.regression.func.X19
gp.fs.0.func.23.nc = nc0
gp.fs.0.func.24 = ec.app.linkmining.regression.func.X20
gp.fs.0.func.24.nc = nc0
gp.fs.0.func.25 = ec.app.linkmining.regression.func.X21
gp.fs.0.func.25.nc = nc0
gp.fs.0.func.26 = ec.app.linkmining.regression.func.X22
gp.fs.0.func.26.nc = nc0
gp.fs.0.func.27 = ec.app.linkmining.regression.func.X23
gp.fs.0.func.27.nc = nc0
gp.fs.0.func.28 = ec.app.linkmining.regression.func.X24
gp.fs.0.func.28.nc = nc0
gp.fs.0.func.29 = ec.app.linkmining.regression.func.X25
gp.fs.0.func.29.nc = nc0
gp.fs.0.func.30 = ec.app.linkmining.regression.func.X26
gp.fs.0.func.30.nc = nc0
gp.fs.0.func.31 = ec.app.linkmining.regression.func.X27
gp.fs.0.func.31.nc = nc0
gp.fs.0.func.32 = ec.app.linkmining.regression.func.X28
gp.fs.0.func.32.nc = nc0
gp.fs.0.func.33 = ec.app.linkmining.regression.func.X29
gp.fs.0.func.33.nc = nc0
gp.fs.0.func.34 = ec.app.linkmining.regression.func.X30
gp.fs.0.func.34.nc = nc0
gp.fs.0.func.35 = ec.app.linkmining.regression.func.X31
gp.fs.0.func.35.nc = nc0
gp.fs.0.func.36 = ec.app.linkmining.regression.func.X32
gp.fs.0.func.36.nc = nc0
gp.fs.0.func.37 = ec.app.linkmining.regression.func.X33
gp.fs.0.func.37.nc = nc0
```

```
gp.fs.0.func.38 = ec.app.linkmining.regression.func.X34
gp.fs.0.func.38.nc = nc0
gp.fs.0.func.39 = ec.app.linkmining.regression.func.X35
gp.fs.0.func.39.nc = nc0
gp.fs.0.func.40 = ec.app.linkmining.regression.func.X36
gp.fs.0.func.40.nc = nc0
gp.fs.0.func.41 = ec.app.linkmining.regression.func.X37
gp.fs.0.func.41.nc = nc0
gp.fs.0.func.42 = ec.app.linkmining.regression.func.X38
gp.fs.0.func.42.nc = nc0
gp.fs.0.func.43 = ec.app.linkmining.regression.func.X39
gp.fs.0.func.43.nc = nc0
gp.fs.0.func.44 = ec.app.linkmining.regression.func.X40
gp.fs.0.func.44.nc = nc0
gp.fs.0.func.45 = ec.app.linkmining.regression.func.X41
gp.fs.0.func.45.nc = nc0
gp.fs.0.func.46 = ec.app.linkmining.regression.func.X42
gp.fs.0.func.46.nc = nc0
gp.fs.0.func.47 = ec.app.linkmining.regression.func.X43
gp.fs.0.func.47.nc = nc0
gp.fs.0.func.48 = ec.app.linkmining.regression.func.X44
gp.fs.0.func.48.nc = nc0
gp.fs.0.func.49 = ec.app.linkmining.regression.func.X45
gp.fs.0.func.49.nc = nc0
gp.fs.0.func.50 = ec.app.linkmining.regression.func.X46
gp.fs.0.func.50.nc = nc0
gp.fs.0.func.51 = ec.app.linkmining.regression.func.X47
gp.fs.0.func.51.nc = nc0
gp.fs.0.func.52 = ec.app.linkmining.regression.func.X48
gp.fs.0.func.52.nc = nc0
gp.fs.0.func.53 = ec.app.linkmining.regression.func.X49
gp.fs.0.func.53.nc = nc0
gp.fs.0.func.54 = ec.app.linkmining.regression.func.X50
gp.fs.0.func.54.nc = nc0
gp.fs.0.func.55 = ec.app.linkmining.regression.func.X51
gp.fs.0.func.55.nc = nc0
gp.fs.0.func.56 = ec.app.linkmining.regression.func.X52
gp.fs.0.func.56.nc = nc0
gp.fs.0.func.57 = ec.app.linkmining.regression.func.X53
gp.fs.0.func.57.nc = nc0
gp.fs.0.func.58 = ec.app.linkmining.regression.func.X54
gp.fs.0.func.58.nc = nc0
gp.fs.0.func.59 = ec.app.linkmining.regression.func.X55
gp.fs.0.func.59.nc = nc0
gp.fs.0.func.60 = ec.app.linkmining.regression.func.X56
gp.fs.0.func.60.nc = nc0
gp.fs.0.func.61 = ec.app.linkmining.regression.func.X57
gp.fs.0.func.61.nc = nc0
gp.fs.0.func.62 = ec.app.linkmining.regression.func.X58
gp.fs.0.func.62.nc = nc0
gp.fs.0.func.63 = ec.app.linkmining.regression.func.X59
gp.fs.0.func.63.nc = nc0
gp.fs.0.func.64 = ec.app.linkmining.regression.func.X60
gp.fs.0.func.64.nc = nc0
gp.fs.0.func.65 = ec.app.linkmining.regression.func.X61
gp.fs.0.func.65.nc = nc0
gp.fs.0.func.66 = ec.app.linkmining.regression.func.X62
```

```
gp.fs.0.func.66.nc = nc0
gp.fs.0.func.67 = ec.app.linkmining.regression.func.X63
gp.fs.0.func.67.nc = nc0
gp.fs.0.func.68 = ec.app.linkmining.regression.func.X64
gp.fs.0.func.68.nc = nc0
gp.fs.0.func.69 = ec.app.linkmining.regression.func.X65
gp.fs.0.func.69.nc = nc0
gp.fs.0.func.70 = ec.app.linkmining.regression.func.X66
gp.fs.0.func.70.nc = nc0
gp.fs.0.func.71 = ec.app.linkmining.regression.func.X67
gp.fs.0.func.71.nc = nc0
gp.fs.0.func.72 = ec.app.linkmining.regression.func.X68
gp.fs.0.func.72.nc = nc0
gp.fs.0.func.73 = ec.app.linkmining.regression.func.X69
gp.fs.0.func.73.nc = nc0
gp.fs.0.func.74 = ec.app.linkmining.regression.func.X70
gp.fs.0.func.74.nc = nc0
gp.fs.0.func.75 = ec.app.linkmining.regression.func.X71
gp.fs.0.func.75.nc = nc0
gp.fs.0.func.76 = ec.app.linkmining.regression.func.X72
gp.fs.0.func.76.nc = nc0
gp.fs.0.func.77 = ec.app.linkmining.regression.func.X73
gp.fs.0.func.77.nc = nc0
gp.fs.0.func.78 = ec.app.linkmining.regression.func.X74
gp.fs.0.func.78.nc = nc0
gp.fs.0.func.79 = ec.app.linkmining.regression.func.X75
gp.fs.0.func.79.nc = nc0
gp.fs.0.func.80 = ec.app.linkmining.regression.func.X76
gp.fs.0.func.80.nc = nc0
gp.fs.0.func.81 = ec.app.linkmining.regression.func.X77
gp.fs.0.func.81.nc = nc0
gp.fs.0.func.82 = ec.app.linkmining.regression.func.X78
gp.fs.0.func.82.nc = nc0
gp.fs.0.func.83 = ec.app.linkmining.regression.func.X79
gp.fs.0.func.83.nc = nc0
gp.fs.0.func.84 = ec.app.linkmining.regression.func.X80
gp.fs.0.func.84.nc = nc0
gp.fs.0.func.85 = ec.app.linkmining.regression.func.X81
gp.fs.0.func.85.nc = nc0
gp.fs.0.func.86 = ec.app.linkmining.regression.func.X82
gp.fs.0.func.86.nc = nc0
gp.fs.0.func.87 = ec.app.linkmining.regression.func.X83
gp.fs.0.func.87.nc = nc0
gp.fs.0.func.88 = ec.app.linkmining.regression.func.X84
gp.fs.0.func.88.nc = nc0
gp.fs.0.func.89 = ec.app.linkmining.regression.func.X85
gp.fs.0.func.89.nc = nc0
gp.fs.0.func.90 = ec.app.linkmining.regression.func.X86
gp.fs.0.func.90.nc = nc0
gp.fs.0.func.91 = ec.app.linkmining.regression.func.X87
gp.fs.0.func.91.nc = nc0
gp.fs.0.func.92 = ec.app.linkmining.regression.func.X88
gp.fs.0.func.92.nc = nc0
gp.fs.0.func.93 = ec.app.linkmining.regression.func.X89
gp.fs.0.func.93.nc = nc0
gp.fs.0.func.94 = ec.app.linkmining.regression.func.X90
gp.fs.0.func.94.nc = nc0
```

```
gp.fs.0.func.95 = ec.app.linkmining.regression.func.X91
gp.fs.0.func.95.nc = nc0
gp.fs.0.func.96 = ec.app.linkmining.regression.func.X92
gp.fs.0.func.96.nc = nc0
gp.fs.0.func.97 = ec.app.linkmining.regression.func.X93
gp.fs.0.func.97.nc = nc0
gp.fs.0.func.98 = ec.app.linkmining.regression.func.X94
gp.fs.0.func.98.nc = nc0
gp.fs.0.func.99 = ec.app.linkmining.regression.func.X95
gp.fs.0.func.99.nc = nc0
gp.fs.0.func.100 = ec.app.linkmining.regression.func.X96
gp.fs.0.func.100.nc = nc0
gp.fs.0.func.101 = ec.app.linkmining.regression.func.X97
gp.fs.0.func.101.nc = nc0
gp.fs.0.func.102 = ec.app.linkmining.regression.func.X98
gp.fs.0.func.102.nc = nc0
gp.fs.0.func.103 = ec.app.linkmining.regression.func.X99
gp.fs.0.func.103.nc = nc0
gp.fs.0.func.104 = ec.app.linkmining.regression.func.X100
gp.fs.0.func.104.nc = nc0
gp.fs.0.func.105 = ec.app.linkmining.regression.func.X101
gp.fs.0.func.105.nc = nc0
gp.fs.0.func.106 = ec.app.linkmining.regression.func.X102
gp.fs.0.func.106.nc = nc0
gp.fs.0.func.107 = ec.app.linkmining.regression.func.X103
gp.fs.0.func.107.nc = nc0
gp.fs.0.func.108 = ec.app.linkmining.regression.func.X104
gp.fs.0.func.108.nc = nc0



%gp.fs.0.func.5 = ec.app.linkmining.regression.func.Sin
%gp.fs.0.func.5.nc = nc1
%gp.fs.0.func.6 = ec.app.linkmining.regression.func.Cos
%gp.fs.0.func.6.nc = nc1
%gp.fs.0.func.7 = ec.app.linkmining.regression.func.Exp
%gp.fs.0.func.7.nc = nc1
%gp.fs.0.func.8 = ec.app.linkmining.regression.func.Log
%gp.fs.0.func.8.nc = nc1


#
# We specify our problem here
#

eval.problem = ec.app.linkmining.regression.Regression
eval.problem.data = ec.app.linkmining.regression.RegressionData
# ADFs use the same data type typically -- we need to include
# this even if we're not implementing ADFs
eval.problem.stack.context.data = ec.app.linkmining.regression.RegressionData

# The size of our training set, by default, is 20
eval.problem.size = 10

pop.subpop.0.size =                 100

stat.file                     $out.stat
```

Appendix C - Symbol Trees for Best Individuals

OneR

1	2	3	4	5
X30	X56	X21	X4	X74
(- X80 X18)	X26	X103	X85	(- X65 X17)
X30	X32	X15	X60	X1
X12	X70	X89	(- X71 X68)	X92
(- X1 X57)	X102	X19	X22	X19
(* X93 X93)	X98	X71	X2	(* X102 X70)
X7	X48	X14	X17	X74
X78	X7	X30	X93	X79
X49	X7	X80	X87	X82
X17	X1	X80	X68	X62

Logistic

1	2	3	4	5
X82	X67	X84	X11	X21
X83	X82	X2	X62	X21
(* X27 X30)	X7	X82	X26	X32
X82	X37	X0	X57	X16
X21	X26	X62	X70	X42
(% X59 X97)	X43	X35	X12	X27
X18	X57	X10	X42	X68
X101	X22	X26	X45	X75
(- X0 X79)	X26	X38	X62	X82
X64	X82	X40	X46	X46

J48

1	2	3	4	5
X37	X18	X27	X78	X78
X21	X101	X57	X86	X94
X84	X21	X28	X5	X21
X92	X58	X86	X7	X97
X4	X68	X42	X18	X43
X5	X82	X82	X46	X0
X78	X91	X21	X48	X9
X46	X42	X78	X23	X31
X103	X21	(* X43 X21)	X102	X3
X27	X75	X48	X78	X4

NaiveBayes

1	2	3	4	5
X70	X23	(% (+ X71 X60) X23)	X4	X14
X82	X18	X92	X11	X23
X19	X82	X43	X43	X82
X26	X0	X21	X25	X12
X38	X21	X2	X21	X75
X82	X51	X60	X22	X18
(% X21 X18)	X43	X18	X26	X26
X38	X54	X34	X82	X0
X47	X104	X42	X30	X44
X28	X95	X79	X81	X73

IB1

1	2	3	4	5
X59	X82	X3	X18	X3
X83	X57	X10	X82	X18
(* X48 X39)	X18	X45	X3	X81
X23	X18	X18	X23	X23
X18	X26	X82	X23	X44
X102	X23	X22	X102	X18
X42	X82	X102	X33	X26
X85	X58	X21	X7	X70
X7	X44	X22	X26	X2
X68	X17	X103	X64	X74