Automated Feature Engineering for Deep Neural Networks with Genetic Programming

by

Jeff Heaton

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

College of Engineering and Computing
Nova Southeastern University

2017

We hereby certify that this dissertation, submitted by Jeff Heaton, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

James D. Cannady, Ph.D. Chairperson of Dissertation Committee	Date
Sumitra Mukherjee, Ph.D. Dissertation Committee Member	Date
Paul Cerkez, Ph.D. Dissertation Committee Member	Date

Approved:

Yong X. Tao, Ph.D., P.E., FASME Dean, College of Engineering and Computing	Date

College of Engineering and Computing
Nova Southeastern University

2017

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Automated Feature Engineering for Deep Neural Networks with Genetic Programming

by
Jeff Heaton
2017

Feature engineering is a process that augments the feature vector of a machine learning model with calculated values that are designed to enhance the accuracy of a model's predictions. Research has shown that the accuracy of models such as deep neural networks, support vector machines, and tree/forest-based algorithms sometimes benefit from feature engineering. Expressions that combine one or more of the original features usually create these engineered features. The choice of the exact structure of an engineered feature is dependent on the type of machine learning model in use. Previous research demonstrated that various model families benefit from different types of engineered feature. Random forests, gradient-boosting machines, or other tree-based models might not see the same accuracy gain that an engineered feature allowed neural networks, generalized linear models, or other dot-product based models to achieve on the same data set.

This dissertation presents a genetic programming-based algorithm that automatically engineers features that increase the accuracy of deep neural networks for some data sets. For a genetic programming algorithm to be effective, it must prioritize the search space and efficiently evaluate what it finds. This dissertation algorithm faced a potential search space composed of all possible mathematical combinations of the original feature vector. Five experiments were designed to guide the search process to efficiently evolve good engineered features. The result of this dissertation is an automated feature engineering (AFE) algorithm that is computationally efficient, even though a neural network is used to evaluate each candidate feature. This approach gave the algorithm a greater opportunity to specifically target deep neural networks in its search for engineered features that improve accuracy. Finally, a sixth experiment empirically demonstrated the degree to which this algorithm improved the accuracy of neural networks on data sets augmented by the algorithm's engineered features.

Acknowledgements

There are several people who supported and encouraged me through my dissertation journey at Nova Southeastern University.

First, I would like to extend my thanks to Dr. James Cannady for serving as my dissertation committee chair. He provided guidance and advice at every stage of the dissertation process. He also gave me several additional research opportunities during my time at NSU. I would also like to thank my committee members Dr. Sumitra Mukherjee and Dr. Paul Cerkez. Their advice and guidance through this process were invaluable.

My sincere gratitude and thanks to Tracy, my wife and graduate school partner. Her help editing this dissertation was invaluable. We took a multi-year graduate journey as she completed her Master's degree in Spanish at Saint Louis University and I worked on my Ph.D. It was great that she could accompany me to Florida for each of the cluster meetings and I could join her in Madrid for summer classes. The GECCO conference that I attended in Madrid was instrumental in the development of this research. Her love and encouragement made this project all possible.

Thank you to Dave Snell for encouraging me to pursue a Ph.D. and for guidance at key points in my career.

I would like to thank my parents. From the time that they bought me my first Commodore 64 and through all my educational journeys, they have always supported me.

Table of Contents

Approval ii
Acknowledgements iv
List of Tables vii
List of Figures viii

1. **Introduction 1**

Problem Statement 6
Dissertation Goal 7
Relevance and Significance 8
Barriers and Issues 9
Definitions of Terms 12
List of Acronyms 22
Summary 23

2. **Literature Review 25**

Feature Engineering 25
Neural Networks 30
Deep Learning 39
Evolutionary Programming 48
Summary 59

3. **Methodology 60**

Introduction 60
Algorithm Contract and Specification 61
Algorithm Design Scope 65
Experimental Design 68
Measures 68
Experiment 1: Limiting the Search Space 70
Experiment 2: Establishing Baseline 73
Experiment 3: Genetic Ensembles 74
Experiment 4: Population Analysis 76
Experiment 5: Objective Function Design 79
Experiment 6: Automated Feature Engineering 81
Real World Data sets 85
Synthetic Data sets 86
Resources 87
Summary 88

4. **Results 90**

Introduction 90
Experiment 1 Results 90
Experiment 2 Results 94
Experiment 3 Results 96

Experiment 4 Results 97
Experiment 5 Results 101
AFE Algorithm Design 103
Experiment 6 Results 116
Summary 123

5. **Conclusions, Implications, Recommendations, and Summary 124**

Conclusions 124
Implications 125
Recommendations 126
Summary 129

References 182

Appendixes

A. **Algorithms 138**
B. **Hyperparameters 144**
C. **Third Party Libraries 152**
D. **Data Set Descriptions 156**
E. **Synthetic Data Set Generation 163**
F. **Engineered Features for the Data Sets 167**
G. **Detailed Experiment Results 173**
H. **Dissertation Source Code Availability 179**

List of Tables

Tables

1. Common neural network transfer functions 41
2. Experiment 1 results format, neural network and genetic programming 72
3. Experiment 2 result format, neural network baseline 74
4. Experiment 3-result format, neural network genetic program ensemble 76
5. Experiment 4 results format, patterns in genetic programs 79
6. Experiment 5 results format, evaluating feature ranking 80
7. Experiment 6 results format, engineered feature effectiveness 84
8. Experiment 1 neural network results for select expressions 92
9. Experiment 1 genetic program results for select expressions 93
10. Experiment 2 baselines for data sets (RMSE) 95
11. Experiment 3, Ensemble, GP & neural network results 97
12. Experiment 4, patterns from genetic programs 100
13. Experiment 6, feature engineered vs non-augmented accuracy 120
14. Experiment 6, T-Test with data sets that showed improvement 122
15. Data sets used by the dissertation experiments 156

List of Figures

Figures

1. Regression and classification network (original features) 3
2. Neural network engineered features 5
3. Elman neural network 35
4. Jordan neural network 35
5. Long short-term memory (LSTM) 37
6. Dropout layer in a neural network 44
7. Expression tree for genetic programming 50
8. Point crossover 52
9. Subtree mutation 53
10. Feature engineering to linearly separate two classes 58
11. Algorithm high level design and contract 64
12. Ensemble of genetic programs for a neural network 75
13. Generate candidate solutions 77
14. Branches with common structures 78
15. High-level overview of AFE algorithm 82
16. Dissertation algorithm evaluation 83
17. Overview of AFE algorithm 105
18. Input neuron weight connections 109
19. Data flow/mapping for the AFE algorithm 112
20. One step of the AFE algorithm 113
21. Collapsing trees 115

22. Experiment 6, feature engineering benefits to data set 121

Chapter 1

Introduction

The research conducted in this dissertation created an algorithm to automatically engineer features that might increase the accuracy of deep neural networks for certain data sets. The research built upon, but did not duplicate, prior published research by the author of this dissertation. In 2008, the Encog Machine Learning Framework was created and includes advanced neural network and genetic programming algorithms (Heaton, 2015). The Encog genetic programming framework introduced an innovative algorithm that allows dynamically generated constant nodes for tree-based genetic programming. Thus, constants in Encog genetic programs can assume any value, rather than choosing from a fixed constant pool.

Research was performed that demonstrated the types of manually engineered features most likely to increase the accuracy of several machine learning models (Heaton, 2016). The research presented here builds upon this earlier research by leveraging the Encog genetic programming framework as a key component of the dissertation algorithm that automatically engineered features for a feedforward neural network that might contain many layers. This type of neural network is commonly referred to as a deep neural network (DNN). Although it would be possible to perform this research with any customizable genetic programming framework or deep neural network framework, Encog is well suited for the task because it provides both components.

This dissertation report begins by introducing both neural networks and feature engineering. The dissertation problem statement is defined, and a clear goal is established. Building upon this goal, the relevance of this study is demonstrated and includes a discussion of the barriers and issues previously encountered by the scientific community. A brief review of literature shows how this research continued previous investigations of deep learning. In addition to the necessary resources and the methods, the research approach to achieve the dissertation goal is outlined.

Most machine learning models, such as neural networks, support vector machines (Smola & Vapnik, 1997), and tree-based models, accept a vector of input data and then output a prediction based on this input. For these models, the inputs are called features, and the complete set of inputs is called a feature vector. Most business applications of neural networks must map the input neurons to columns in a database; these inputs allow the neural network to make a prediction. For example, an insurance company might use columns for age, income, height, weight, high-density lipoprotein (HDL) cholesterol, low-density lipoprotein (LDL) cholesterol, and triglyceride level (TGL) to make suggestions about an insurance applicant (B. F. Brown, 1998).

Inputs such as HDL, LDL, and TGL are all named quantities. This can be contrasted to high-dimensional inputs such as pixels, audio samples, and some time-series data. For consistency, this dissertation report will refer to lower-dimensional data set features that have specific names as *named features*. This dissertation focused upon such named features. High-dimensional inputs that do not assign specific meaning to individual features fell outside the scope of this research.

Classification and regression are the two most common applications of neural networks. Regression networks predict a number, whereas classification networks assign a non-numeric class. For example, the maximum policy face amount is the maximum amount that the regression neural network suggests for an individual. This is a dollar amount, such as $100,000. Similarly, a classification neural network can suggest the non-numeric underwriting class for an individual, such as preferred, standard, substandard, or decline. Figure 1 shows both types of neural network.

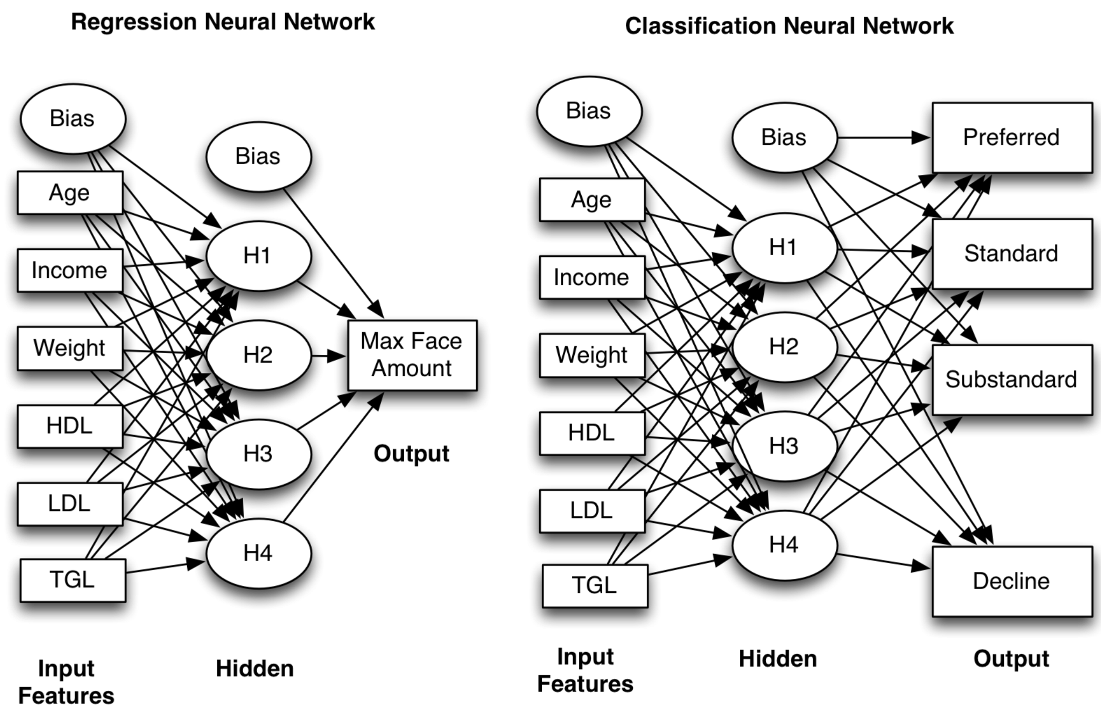


Figure 1. Regression and classification network (original features)

The left neural network performs a regression and uses the six original input features to set the maximum policy face amount to issue an applicant. The right neural network executes a classification and utilizes the same six input features to place the insured into an underwriting class. The weights (shown as arrows) establish the final output. A backpropagation algorithm fixes the weights through many sets of inputs that all have a known output. In this way, the neural network learns from existing data to predict future data. Furthermore, for simplicity, the above networks have a single hidden layer. Deep neural networks typically have more than two hidden layers between the input and output layers (Bengio, 2009). Every layer except the output layer can also receive a bias neuron that always outputs a consistent value (commonly 1.0). Bias neurons enhance the neural network's learning ability (B. Cheng & Titterington, 1994).

Output neurons provide the neural network's numeric result. Before the output neurons can be determined, the values of previous neurons must be calculated. The following equation can determine the value of each neuron:

$$\boldsymbol{f}(\boldsymbol{x},\boldsymbol{\theta},\boldsymbol{b}) = \boldsymbol{\phi}\left(\sum\nolimits_{\boldsymbol{i}}(\boldsymbol{\theta}_{\boldsymbol{i}}\boldsymbol{x}_{\boldsymbol{i}}) + \boldsymbol{b}\right) \tag{1}$$

The function *phi* (ϕ) represents the transfer function, and it is typically either a rectified linear unit (ReLU) or one of the sigmoidal functions. The vectors θ and x represent the weights and input; the variable b represents the bias weight. Calculating the weighted sum of the input vector (x) is the same as taking the dot product of the two vectors. This calculation is why neural networks are often considered to be part of a larger class of machine learning algorithms that are dot-product based.

For input neurons, the vector x comes directly from the data set. Now that the input neuron values are known, the first hidden layer can be calculated, using the input neurons as x. Each subsequent layer is calculated with the previous layer's output as x. Ultimately, the output neurons are determined, and the process is complete.

Feature engineering adds calculated features to the input vector (Guyon, Gunn, Nikravesh, & Zadeh, 2008). It is possible to use feature engineering for both classification and regression neural networks. Engineered features are essentially calculated fields that are dependent on the other fields. Calculated fields are common in business applications and can help human users understand the interaction of several fields in the original data set. For example, insurance underwriters benefit from combining height and weight to calculate body mass index (BMI). Likewise, insurance underwriters often use a ratio of the HDL, TGL and LDL cholesterol levels. These calculations allow a single number to represent an aspect of the health of the applicant. These calculations might also be useful to the neural network. If the BMI and HDL/LDL ratios were engineered as features, the classification network would look like Figure 2.

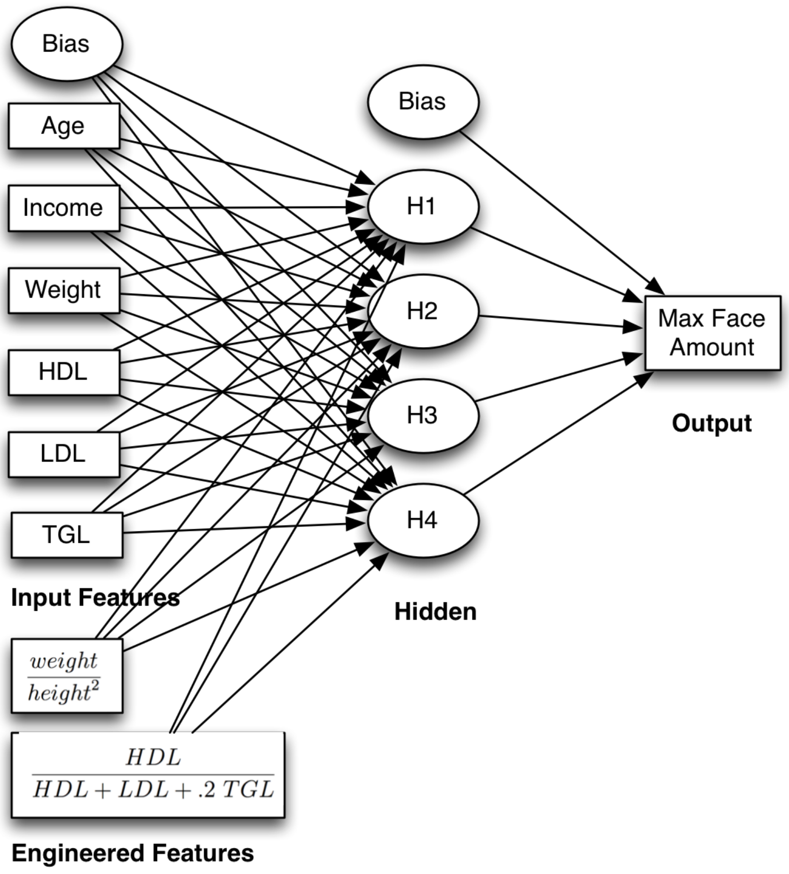


Figure 2. Neural network engineered features

In Figure 2, the BMI and HDL/LDL ratio values are appended to the feature vector along with the original input features. This calculation produces an augmented feature vector that is provided to the neural network. These additional features might help the neural network to calculate the maximum face amount of a life insurance policy.

Similarly, these two features could also augment the feature vector of a classification neural network. BMI and the HDL/LDL ratio are typical of the types of features that might be engineered for a neural network. Such features are often ratios, summations, and powers of other features. Adding BMI and the HDL/LDL ratio is not complicated because these are well-known calculations. Similar calculations might also benefit other data sets. Feature engineering often involves combining original features with ratios, summations, differences, and power functions. BMI is a type of engineered feature that involves multiple original features and is derived manually using intuition about the data set.

Problem Statement

There is currently no automated means of engineering features specifically designed for deep neural networks that are a combination of multiple named features from the original vector. Previous automated feature engineering (AFE) work focused primarily upon the transformation of a single feature or upon models other than deep learning (Box & Cox, 1964; Breiman & Friedman, 1985; Freeman & Tukey, 1950). Although model-agnostic genetic programming-based feature extraction algorithms have been proposed (Guo, Jack, & Nandi, 2005; Neshatian, 2010), they do not tailor engineered features to deep neural networks. Feature engineering research for deep learning has primarily dealt with high-dimensional image and audio data (Blei, Ng, & Jordan, 2003; M. Brown & Lowe, 2003; Coates, Lee, & Ng, 2011; Coates & Ng, 2012; Le, 2013; Lowe, 1999; Scott & Matwin, 1999).

Machine learning performance depends on the representation of the feature vector. Feature engineering is an important but labor-intensive component of machine learning applications (Bengio, 2013). As a result, much of the actual effort in deploying machine learning algorithms goes into the design of preprocessing pipelines and data transformations (Bengio, 2013).

Deep neural networks (Hinton, Osindero, & Teh, 2006) can benefit from feature engineering. Most research into feature engineering in the deep learning field has been in the areas of image and speech recognition (Bengio, 2013). Such techniques are successful in the high-dimension space of image processing and often amount to dimensionality reduction techniques (Hinton & Salakhutdinov, 2006) such as Principal Component Analysis (PCA) (Timmerman, 2003) and auto-encoders (Olshausen & Field, 1996).

Dissertation Goal

The goal of this dissertation was to use genetic programming to analyze a data set of named features and to automatically create engineered features that would produce a more accurate deep neural network. These engineered features consisted of mathematical transformations of one or more of the existing features from the data set. Feature engineering will only improve accuracy when the engineered feature exposes an interaction that the neural network could not determine from the data set (W. Cheng, Kasneci, Graepel, Stern, & Herbrich, 2011). Consequently, feature engineering will not benefit all data sets. To mitigate this problem, several real-world data sets, as well as synthetic data sets designed specifically to benefit from feature engineering, were evaluated.

The research focused on data sets consisting of named features, as opposed to data sets that contain large amounts of pixels or audio sampling data. Fraud monitoring, sales forecasting, and intrusion detection are predictive modeling applications in which the input is composed of these named values. Real-world data sets that provide a basis for predictive modeling can often consist of fewer than 100 such named features. Therefore, the algorithm developed would be ineffective for the high-dimension tasks of computer vision and hearing applications. Such data sets were outside of the scope of this dissertation research.

Relevance and Significance

Since the introduction of multiple linear regression, statisticians have been employing creative means to transform input to enhance model accuracy (Anscombe & Tukey, 1963; Stigler, 1986). These transformations usually applied a single mathematical expression to an individual feature. For example, one feature might apply a logarithm; another feature might be raised to the third power. Transformations that apply an expression to a single original feature can significantly increase the accuracy of certain model types (Kuhn & Johnson, 2013).

Researchers have focused much attention on automated derivation of single-feature transformations. Freeman and Tukey (1950) reported on a number of useful transformations for linear regression. Box and Cox (1964) conducted the seminal work on automated feature transformation and invented a stochastic ad hoc algorithm that recommends transformations that might improve the results of linear regression. Their work became known as the Box-Cox transformation. Although the work by Box and Cox can obtain favorable transformations for linear regression, it often did not converge to the best one for an individual feature because of its stochastic nature. Numerous other transformations were created that were based on similar stochastic sampling techniques (Anscombe & Tukey, 1963; Mosteller & Tukey, 1977; Tukey, Laurner, & Siegel, 1982).

Each of these algorithms focused on transformations of single features for the linear regression model.

Breiman and Friedman (1985) took a considerable step forward by introducing the alternating conditional expectation (ACE) algorithm that guaranteed optimal transformations for linear regression. Even though all the aforementioned algorithms were designed for linear regression, they can also assist other model types. Because ACE was designed for linear regression, it cannot guarantee optimal transformations for other model types (Ziehe, Kawanabe, Harmeling, & Müller, 2001). Additionally, ACE transforms the entire feature vector by using separate transformations for each input feature, as well as the output. Engineered features do not need to come from the transformation of a single feature from the original vector; several original input features can be transformed collectively to produce favorable results (H.-F. Yu et al., 2011).

Feature engineering played an important role in several winning Kaggle and ACM's KDD Cup submissions. H.-F. Yu et al. (2011) reported on the successful application of feature engineering to the KDD Cup 2010 competition. Ildefons and Sugiyama (2013) won the Kaggle Algorithmic Trading Challenge with an ensemble of models and feature engineering. A manual process created the features engineered for these competitions.

Barriers and Issues

It was a difficult and challenging problem to create an algorithm that would automatically engineer features for a deep neural network. Most AFE algorithms are built for linear regression models. Furthermore, most of these automated methods typically engineer features that are based upon only a single feature from the original vector. The few AFE algorithms that are compatible with neural networks often do not make use of an actual neural network or do not exploit characteristics unique to neural networks. Neural networks are often excluded from the objective function because neural networks are computationally expensive. The unique characteristics of neural networks can be exploited by tailoring the search space to expression types that neural networks cannot easily synthesize on their own.

A linear regression model is a multi-term, first-degree expression. Because of its simplicity, several mathematical techniques can verify and assure optimal transformations (Breiman & Friedman, 1985). Deep neural networks have a much more complex structure than linear regression. Therefore, engineering features for a deep neural network present different challenges than those for a linear regression.

This dissertation created an algorithm that combined multiple original features. As the number of original features considered increases, so does the search space of the algorithm. This expanded search space faced the curse of dimensionality (Bellman, 1957) and required novel solutions to limit and prioritize the search space. However, ignoring the curse of dimensionality in favor of a simple theoretical approach to AFE can generate every possible expression based on the original feature set. Obviously, this is a large task. Even with a small number of features and one expression type, the search space is still large. As an example, consider searching every combination of an engineered rational difference as demonstrated by the following expression:

$$\frac{\boldsymbol{a-b}}{\boldsymbol{c-d}} \tag{2}$$

If the data set were modestly large and contained 100 features in the original data set, the complete search space would be over 47 million permutations:

$$\mathbf{0.5\,P(100,4) = 0.5\frac{100!}{(100-4)!} = 4.705\ \times 10^{7}} \tag{3}$$

The above permutation is divided by two to account for the fact that the resulting expression is algebraically the same as the first if the order of both the numerator and denominator's differences are flipped. This example is only one of the many types of features that the algorithm needed to check. These expressions were prioritized so that the most likely successful candidates are explored first. It is possible to prioritize candidate features for evaluation by the analysis of the importance of the original features.

Although it was necessary to reduce the search space, finding an efficient means of evaluating candidate feature transformations was also crucial. An obvious test of a feature transformation would have been to fit a neural network with the candidate feature and then fit a neural network without it. However, neural networks start from random weights, so it is advisable to fit several times and calculate the lowest error across those runs to mitigate the effects of a bad set of starting weights. It might have also been advantageous to try several neural network hidden weight architectures to ensure that a candidate-engineered feature was not simply reacting badly to a poorly architected neural network. Obviously, it was not possible to provide such an exhaustive evaluation for each candidate engineered feature. The evaluation code needed to be quick and able to run in parallel. Additionally, the neural networks had to be structured so that they gave the best assessment possible for each candidate-engineered feature. These problems had to be overcome to give the best possible ranking of the candidate-engineered features.

The fact that not all data sets benefit from feature engineering created another complication. As a result, it was necessary to use a variety of data sets from the UCI Machine Learning Repository (Newman & Merz, 1998). It was also essential to generate several synthetic data sets that were designed to benefit from certain types of engineered features. Considerable experimentation was required to try all selected real and synthetic data sets against the dissertation algorithm.

Definitions of Terms

Automated Feature Engineering (AFE): The process where an algorithm automatically performs the normally manual task of feature engineering.

Adaptive Moment Estimation (Adam): A training update rule that automatically scales individual learning rates for each weight, deals well with sparse gradients, and can handle a stochastic objective function.

Alternating Condition Expectation (ACE): A form of automatic feature transformation that is applied to both the input and output of a linear regression model.

Auto-Encoder: A neural network that can learn efficient encodings for data to perform dimension reduction. An auto-encoder will always have the same number of input and output neurons.

Bias Neuron: A neuron that is added to the input and hidden layers that always applies a fixed weight. The bias neuron performs a similar function as the *y*-intercept in linear regression.

Box-Cox Transformation: An automatic feature transformation for linear regression that can find helpful transformations for individual features.

Classification: A neural network, or other model, that is trained to classify its input into a predefined number of classes. A classification neural network will typically use a softmax function on its output and have the same number of output neurons as the total number of classes.

Constant Node: A terminal node in a genetic program that holds a constant value.

Constant Pool: A fixed-length pool of numbers that can be assigned to constant nodes in a genetic program.

Cross Entropy Loss Function: A neural network loss function that often provides superior training results for classification when compared to the quadratic loss function. For regression problems, root mean square error (RMSE) should be considered.

Crossover: An evolutionary operator, inspired by the biological concept of sexual reproduction in which two genomes combine traits to produce offspring. Crossover performs the exploitation function of an evolutionary algorithm by creating new genomes based on fit parents.

Data Set: For supervised training, a data set is a collection of input values (*x*) and the expected output values (*y*). The presented research only deals with supervised training. The overall data set is usually divided into training and validation data sets.

Deep Learning: A collection of training techniques and neural network architecture innovations that make it possible to effectively train neural networks with three or more layers.

Deep Neural Network: A neural network with three or more layers.

Directed Acyclic Graph (DAG): A graph where all connections contain directions and there are no loops. The genetic programs found in this research are represented as DAGs.

Dot Product Based Model: A model that uses dot products, or weighted sums, as a primary component of their calculation. Neural networks, linear regressions, and support vector machines for regression are all examples of dot product based models.

Dynamic Constant Node: A special type of constant node found in Encog genetic programs that can change its value as genetic programming training progresses. This is different than the constant pool found in many genetic programming frameworks.

Elitism: When a certain number of the top genomes of a population are chosen to always survive into the next generation.

Engineered Feature: A feature that is added to the feature vector of a model, such as a neural network, that is a mathematical transformation of one or more features from the original feature vector.

Epoch: A unit in an iterative training algorithm where the entire training set has been processed.

Evolutionary Algorithm: An optimization algorithm that evolves better-suited individuals from a population by applying mutation and crossover. The algorithm must balance between exploring new solutions and exploiting existing solutions to make them better.

Expression: A mathematical representation that involves operators, constants, and variables. Expressions were the genomes that the algorithm manipulated in this dissertation.

Exploitation: The process in a search algorithm where the evaluation is constrained to regions close to the best solution discovered so far.

Exploration: The process in a search algorithm where the search is broadened to new regions farther away from the best solution discovered so far.

Feature Engineering: The process of creating new features by applying mathematical transformations to one or more of the original features.

Feature Importance: A measurement of the importance of a single feature to the neural network, relative to the other features.

Feature Selection: The process of choosing the most important features to the neural network.

Feature Vector: The complete set of inputs to a neural network. The feature vector must be the same size as the input layer.

Feature: A single value from the feature vector of a neural network or another model. The features in a feature vector are directly connected to the input neurons of a neural network.

Feedforward Neural Network: A neural network that contains only forward connections between the layers.

Gated Recurrent Unit (GRU): A computationally efficient form of Long Short-Term Memory (LSTM).

Genetic Programming (GP): An evolutionary algorithm that develops programs to optimize their output for an objective function.

Genome: An individual in an evolutionary program.

Gradient: A partial derivative of the loss function of a neural network with respect to a single weight. The gradient is a key component in many neural network-training algorithms.

Hidden Layer: A neural network layer that occurs between the input and output layers. A neural network can contain zero or more hidden layers.

Input Layer: The first layer of neurons that receives the input to the neural network. Neural networks have a single input layer that must be the same length as the feature vector to the neural network.

Iteration: One unit of work in an iterative algorithm. If an iteration only processes a batch that is not the entire training set, then the iteration is called a step. A series of steps that process an entire data set are called an epoch.

Interior Node: A node in a genetic program that is not a terminal node or root node; it has both parents and children.

Kaggle: A website where individuals compete to achieve the best model for posted data sets. Kaggle is considered as a source of benchmarking data sets in this research.

KDD Cup: An annual competition, hosted by the Association for Computing Machinery (ACM) where individuals compete for the best model fit on a provided data set.

Latent Dirichlet Allocation (LDA): A natural language processing generative statistical model.

Layer: A collection of related neurons in a neural network. A neural network typically has an input, output, and zero or more hidden layers.

Learning Rate: A numeric constant that controls how quickly a model, such as a neural network, learns. For backpropagation, the learning rate is multiplied by the gradient to determine the value to change the weight.

Linear Discriminant Analysis (LDA): Generalization of the classification algorithm originally proposed by Fisher for the iris data set.

Linear Regression: A simple model that computes its output as the weighted sum of the input plus an intercept value.

Local Optima: A potential solution to an optimization problem that has no better solutions nearby in the search space. This local solution can prevent an optimization algorithm from finding other better solutions. These solutions are sometimes called local minima or local maxima, depending on if the optimization algorithm is seeking minimization or maximization.

Long Short-Term Memory (LSTM): A type of recurrent neural network that uses a series of gates to learn patterns spanning much larger sequences than those that regular simple recurrent networks are capable of learning.

Loss Function: A function that measures the degree to which the actual input from a neural network ($\hat{y}$) is different than the expected output (y).

Momentum: A numeric constant that attempts to prevent a neural network from falling into a local minimum. This value is multiplied by the previous iteration's weight change to determine a value to add to the current iteration's weight change.

Mutation: An evolutionary operator, inspired by the biological concept of asexual reproduction, in which a single genome produces offspring with a slightly altered set of traits than the single parent. Because mutation introduces new stochastic information, it fulfills the exploration component of an evolutionary algorithm.

Named Feature: A description used in this dissertation for a feature that represents a specific value, such as a measurement or a characteristic of the data. An image is a high-dimension input that would contain pixels, as opposed to named features.

Natural Language Processing: Artificial intelligence algorithms that are designed to understand human language.

Nesterov Momentum: A more advanced form of classic momentum that attempts to mitigate the effects of over correcting to a bad training batch.

Neural Network: A model inspired by the human brain that is composed of an input layer, zero or more hidden layers, and an output layer.

Moment: Measure of the points in a probability density function. The zeroth moment is the total probability (i.e. one); the first moment is the mean; the second central moment is the variance; the third moment is the skewness; the fourth moment (with normalization and shift) is the kurtosis. The ADAM training algorithm, used in this dissertation, estimates the first and second moments of the gradients.

Objective Function: A function that evaluates a genetic program (genome) and produces a score. The evolutionary algorithm will try to create genomes that either maximize or minimize this score. Most evolutionary algorithms can be configured to either maximize or minimize.

One-Versus-Rest: A technique that allows multiple binary classification models to perform multiclass classification by training one classification model per class to classify between that class and the rest of the classes.

Output Layer: The final layer of a neural network. This layer produces the output. A regression neural network will typically have a single output neuron. A binary classification neural network will also have a single output neuron. A classification neural network with three or more classes will have an output neuron for each class.

Preprocessing: An algorithm that prepares data for a neural network or another model. The feature engineering explored in this paper would function as a part of data preprocessing for a neural network.

Principal Component Analysis (PCA): A form of dimension reduction that can shrink the size of an input vector to a smaller encoding with minimal loss of accuracy to the neural network.

Quadratic Loss Function: A simple neural network loss function that uses the difference between the expected output and actual output of a neural network. The quadratic loss function should be the first choice for regression neural networks; however, the cross-entropy loss function should be the choice for classification neural networks.

Recurrent Neural Network: A neural network that contains backwards connections from layers to previous layers.

Regression: A neural network or other model that is trained to produce a continuous value as its output. A regression neural network will use a linear transfer function on its output and have a single output neuron.

Root Mean Square Error (RMSE) or Root Mean Square Deviation (RMSD): A neural network loss function that reports error in the same units as the data set target values. Although RMSE/RMSD is more common in regression problems, it can be used to measure the accuracy of the probabilities reported by classification problems. RMSE is the primary evaluation method used in this dissertation.

Root Node: The node in a tree that is an ancestor of all other nodes. The root node for a single-node tree is also a terminal node.

Selection: An algorithm that chooses fit genomes for evolutionary operations such as crossover and mutation.

Sigmoidal: Something that is s-shaped.

Simple Recurrent Network (SRN): A network with only a single recurrent connection, such as an Elman or Jordan network.

Softmax: An algorithm that ensures that all outputs of a neural network sum to 1.0, thereby allowing the outputs to be considered as probabilities.

Step: A unit of work in an iterative training algorithm. A step does not process the entire training set, only a batch.

Stochastic Gradient Descent (SGD): A variant of the backpropagation algorithm that uses a mini-batch that is randomly sampled each training iteration. SCG has proven itself to be one of the most effective training algorithms, and it was the neural network training method for this dissertation, along with the Adam update rule.

Symbolic Expression (S-Expression): Notation for nested list (tree-structured) data, invented for the Lisp programming language, which uses it for source code as well as data.

Synthetic Data set: A data set that was generated to test a specific characteristic of an algorithm.

Terminal Node: A node in a tree that has no children. Terminal nodes are also referred to as leaf nodes.

Training Data set: The data on which the model was trained. Usually, validation data are also kept so that the model can be evaluated on different data than it was trained with.

Transfer Function: Applied to the weighted summations performed by the layers of a neural network. All layers of a neural network have transfer functions except the input layer. Transfer functions are sometimes referred to as activation functions.

Tree-Based Genetic Program: A genetic program that is represented as a tree of nodes. The research performed by this dissertation used tree-based genetic programs.

Turing Complete: A system of data-manipulation rules that simulates any single-taped Turing machine. Also, referred to as computationally universal.

Universal Approximation Theorem: A theorem that states that a feedforward network with a single hidden layer containing a finite number of neurons can approximate continuous functions.

Update Rule: The method by which a training algorithm updates the weights of a neural network. Common update rules include: Momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, and Adam.

Validation Data set: The portion of the data set that validates model predictions on data that are outside of the training data set. This data set is sometimes referred to as out-of-sample data.

Xavier Weight Initialization: A neural network weight initialization algorithm that produces relatively quick convergence for backpropagation and limited variance of required iteration counts for repeated training of a neural network.

YAML Ain't [sic] Markup Language (YAML): A common configuration file format that communicates operating parameters to the algorithm provided by this research.

List of Acronyms

ACE: Alternating Condition Expectation

AFE: Automated Feature Engineering

ANN: Artificial Neural Network

CGP: Cartesian Genetic Programming

DAG: Directed Acyclic Graph

DBNN: Deep Belief Neural Network

DNN: Deep Neural Network

GP: Genetic Programming

GRU: Gated Recurrent Unit

IDS: Intrusion Detection System

KDD: Knowledge Discovery in Databases

LDA: Latent Dirichlet Allocation or Linear Discriminant Analysis

LSTM: Long Short-Term Memory

NEAT: NeuroEvolution of Augmenting Topologies

ReLU: Rectified Linear Unit

RDBMS: Relational Database Management System

SD: Standard Deviation

SGD: Stochastic Gradient Descent

SRN: Simple Recurrent Neural Network

SIFT: Scale-Invariant Feature Transform

TD-IDF: Term Frequency–Inverse Document Frequency

T-SNE: t-Distributed Stochastic Neighbor Embedding

XOR: Exclusive Or

YAML: YAML Ain't [sic] Markup Language

Summary

This chapter introduced research that was conducted into AFE. The resulting algorithm leveraged the ability of genetic programming to generate expressions that might become useful features for neural networks. The features engineered by this algorithm consisted of expressions that utilized one or more features from the data set’s original feature vector. The resulting engineered features were demonstrated to increase the accuracy of the neural network for some data sets.

The primary challenge of this research was to limit the large search space of expressions that combined the features of the data set. This goal was accomplished by defining the types of expressions that most benefitted a neural network, determining the importance of a feature, and prioritizing the expressions to be searched by the algorithm. The algorithm was also designed to run in parallel to utilize multiple cores on the host computer system.

Additionally, the algorithm was designed to include an efficient objective function to evaluate candidate-engineered features against each other. Such a function was based on fitting a neural network model with candidate-engineered features. This objective function was designed to be efficient so that it could determine in minimal time how effective one candidate engineered feature was compared to another. Genetic programming uses this type of objective function to decide the best genomes (candidate-engineered features) to form the next generation of genomes.

The remainder of this dissertation report is organized as follows: Chapter 2 provides a review of the literature that directly influenced this research. Chapter 3 presents the methodology applied to implement an algorithm for AFE. Chapter 4 summarizes the results of this research. Chapter 5 offers up conclusions based on this research.

Chapter 2

Literature Review

The research conducted for this dissertation focused primarily upon feature engineering and how to apply it to deep neural networks. Because of its ability to manipulate expressions, genetic programming enabled the dissertation algorithm to recommend engineered features. The following areas of literature were important to this dissertation research:

- Feature engineering
- Neural networks
- Deep learning
- Genetic programming

There is considerable research community interest in these areas. The following sections review current literature in these areas as it pertains to the dissertation research:

Feature Engineering

The feature vector of a predictive model can be augmented or transformed to enhance predictive performance. In literature, this process is often referred to as feature engineering, feature modification, or feature extraction. Automated variants of these processes are sometimes referred to as automated feature engineering, automated feature engineering, or feature learning. For consistency, this dissertation report will use the term *feature engineering* to refer to this augmentation. This report will use *automated feature engineering (AFE)* to refer to an algorithm that automates this feature engineering.

The Box and Cox (1964) algorithm relied upon a stochastic sampling of the data and does not necessarily guarantee an optimal set of transformations. Breiman and Friedman (1985) introduced the alternating conditional expectation (ACE) algorithm that could ensure optimal transformations for linear regression. The ACE algorithm finds a set of optimal transformations for each of the predictor features, as well as the output for linear regression. Although the resulting transformations were originally intended for linear regression, they work for other model types as well (B. Cheng & Titterington, 1994).

Splines are a common means of feature transformation for most machine learning model types. By fitting a spline to individual features, it is possible to smooth the data and reduce overfitting. The number and position of knots inside the spline is a hyper-parameter that must be determined for this transformation technique. Splines have the capability of taking on close approximations of the shape of other functions. Brosse, Lek, and Dauba (1999) used splines to transform data for a neural network to predict the distribution of fish populations.

Machine vision has been a popular application of feature engineering. A relatively early form of feature engineering for computer vision was the Scale-Invariant Feature Transform (SIFT) (Lowe, 1999). This transformation attempts the recognition of images at multiple scales, which is a common problem in computer vision. A machine learning model that learns to recognize digits might not perceive these same digits if their size is doubled. SIFT preprocesses the data and provides them in a form where images at multiple scales produce features that are similar. These types of features can be generalized for many problems and applications in machine vision, including object recognition and panorama stitching (M. Brown & Lowe, 2003).

Text classification is another popular application of machine learning algorithms. Scott and Matwin (1999) utilized feature engineering to enhance the performance of rules learning for text classification. These transformations allow structure and frequency of the text to be generalized to a few features. Representing textual data to a machine learning model produces a considerable number of dimensions. Text classification commonly uses feature engineering to reduce these dimensions.

Another application of feature engineering to text classification is the latent Dirichlet allocation (LDA) engineered feature. This method transforms a corpus of documents into document-topic mappings (Blei et al., 2003). LDA has subsequently been applied to spam filtering, among several document classification tasks (Bíró, Szabó, & Benczúr, 2008) and article recommendation (C. Wang & Blei, 2011).

Many data are stored in relational database management systems (RDBMS) and consist of several different tables that form links, or relations, between them. The relationships between these tables can be of various cardinalities, leading to relationships including one-to-one, one-to-many, or many-to-many. Machine learning models typically require a fixed-length feature vector. Mapping the RDBMS linked data into a feature vector that is suitable for a machine learning model can be difficult. Automated mapping of RDBMS data is an active area of research. Bizer, Heath, and Berners-Lee (2009) created a system where the data are structured in a way that they can be accessed with semantic queries.

Feature engineering has proven to be valuable in the Kaggle and KDD Cup data science competitions. In fact, one team utilized feature engineering and an ensemble of machine learning models to win the KDD Cup 2010 competition (H.-F. Yu et al., 2011).

Histograms of oriented gradients were other features presented in this competition. W. Cheng et al. (2011) developed an automated feature generation algorithm for data organized with domain-specific knowledge. These technologies have found many applications. For example, Ildefons and Sugiyama (2013) were able to win the Kaggle Algorithmic Trading Challenge with an ensemble of models and feature engineering. The features engineered for these competitions were created manually or with knowledge about the specific competition problem. Such knowledge is referred to as domain-specific knowledge and requires human intuition that cannot currently be replicated by a machine.

Feature engineering has also advanced natural language processing (NLP). An example of an engineered feature for NLP is the term frequency inverse document frequency (TF-IDF). This engineered feature is essentially the ratio of the frequency of a word in a document compared to its occurrence in the corpus of documents (Rajaraman & Ullman, 2011). TF-IDF has proven popular for text mining, text classification, and NLP.

Researchers have examined the ability of machine learning algorithms to perform automated feature learning. These algorithms are often unsupervised in that they examine the data without regard to an expected outcome. Coates et al. (2011) implemented an unsupervised single-layer neural network for feature engineering. Principal component analysis (PCA) (Timmerman, 2003) and t-distributed stochastic neighbor embedding (T-SNE) (Van der Maaten & Hinton, 2008) are dimension-reduction algorithms that have also proven to be successful for AFE. Other unsupervised machine learning algorithms have also been applied to feature engineering. Coates and Ng (2012) utilized k-means clustering for feature engineering.

Deep neural networks have many different layers to learn complex interactions in the data. Despite this advanced learning capability, deep learning also benefits from feature engineering. Bengio (2013) demonstrated that feature engineering is useful for speech recognition, computer vision, classification, and signal processing. Le (2013) engineered high-level features using unsupervised techniques to construct a deep neural network for signal processing.

Lloyd, Duvenaud, Grosse, Tenenbaum, and Ghahramani (2014) employed feature engineering to create the *Automatic Statistician* project. This system spontaneously models regression problems and produces readable reports. This system can determine the types of transformations that might benefit individual features.

Kanter and Veeramachaneni (2015) invented a technique called deep feature synthesis that automatically transforms relational database tables into the feature vector needed by the typical machine learning model. The feature vector input to a neural network cannot directly encode the one-to-many and many-to-many relationships that are common in RDBMSs. The deep feature synthesis algorithm employs SQL-like transformations, such as MIN, MAX, and COUNT to summarize and encode these relationships into a feature vector. The authors of deep feature synthesis reported on their algorithm’s ability to outperform some competitors in three data science competitions.

Researchers have implemented AFE for specific domains. Davis and Foo (2016) applied automated feature detection to modeling tasks involving HTTP traffic and tunnels. Cuayáhuitl (2016) created SimpleDS, a system for text document processing that avoids manual feature engineering by using a deep reinforcement learning system. Manual feature engineering remains popular, and researchers have explored its value in various contexts. Bahnsen, Aouada, Stojanovic, and Ottersten (2016) provide guidelines and examples of feature engineering for credit card fraud detection. This application is an example of domain-specific knowledge. Zhang, Huan, and Jiang (2016) investigated feature engineering for phishing detection. Although these systems are effective in their specific domains, they do not address the problem statement given by this dissertation research of creating a generic AFE system.

Neural Networks

Neural networks are a biologically-inspired class of algorithms that McCulloch and Pitts (1943) introduced as networks composed of MP-Units. Although neural networks contain connections and neurons, they do not attempt to emulate every aspect of biological neurons. Modern neural networks are more of a mathematical model than a biological simulator. The seminal neural network algorithm of McCulloch and Pitts specifies the calculation of a single neuron, called an MP-Unit, as the weighted sum of the neuron's inputs. This weighted sum is a mathematical dot product. Nearly all neural networks created since their introduction in 1943 are based upon feeding dot product calculations to transfer functions over layers of neurons. Deep neural networks simply have more layers of neurons (MP-Units).

Initially, the weights of neural networks were handcrafted to create networks capable of solving simple problems. The research community has shown great interest in automating neural network weight selection to achieve a particular objective. Hebb (1949) defined a process to describe how the connection strengths between biological neurons change as learning occurs. When the organism performs actions, connections increase between the neurons necessary for that action. This process became Hebb's rule, and it is often informally stated as, "neurons that fire together wire together."

Rosenblatt (1962) introduced the perceptron that became the seminal neural network that contained input and output layers. The perceptron is a two-layer neural network with an input layer that contains weighted forward-only connections to an output layer. The transfer function defined for the classic perceptron is a simple function that performs a threshold—it returns the value 1 if the neuron's weighted inputs reach a value above a specified threshold; otherwise, it returns the value 0. Minsky and Papert (1969) described severe limitations in the perceptron in their monograph. They demonstrated that perceptrons were incapable of learning the Exclusive Or (XOR) operator, a non-linearly separable problem.

Research continued for the automatic derivation of the weights of a neural network, beginning with the work of Werbos (1974) when he mentioned that gradient descent could be used for the training of neural networks in his Ph.D. thesis. Previously, researchers only used gradient descent to find minimums of functions. Rumelhart, Hinton, and Williams (1985) were the first to apply gradient descent to neural network training. Their algorithm is called the backward propagation of errors, or backpropagation. The gradient of each weight is calculated and determines a change that should occur in the weight for the current training iteration. The gradient of each weight is the partial derivative of the loss function for that weight with all other weights held constant. Therefore, backpropagation applies gradient descent to neural network training. Standard backpropagation computes a weight correction (v) for the iteration (t) that minimizes the error function (*J*) by updating previous iteration's weight (θ_{t-1}) by the weight's gradient scaled by the learning rate (η):

$$\boldsymbol{v_t = -\eta \cdot \nabla_{\theta_{t-1}} J(\theta_{t-1})} \tag{4}$$

The weight correction (v_t) is applied to the previous neural network weight (θ_{t-1}) to give the weight (θ_t) for the current iteration:

$$\boldsymbol{\theta_t = \theta_{t-1} + v_t} \tag{5}$$

Backpropagation was initially ineffective at training neural networks with significantly more than two hidden layers (Bengio, 2009). Furthermore, it was not known if neural networks actually benefited from many layers. Gybenko (1989) formulated the universal approximation theorem and proved that a single hidden-layer neural network could approximate any function. Hornik (1991) continued this research by showing that the multilayer feedforward architecture – and not the specific choice of the transfer function – gave neural networks the potential of being universal approximators. The universal approximation theorem implies that additional hidden layers are unnecessary because a single hidden-layer neural network can theoretically learn any problem. Although feedforward neural networks are universal approximators, they are not Turing complete (Graves, Wayne, & Danihelka, 2014; Turing, 1936) without extension. In other words, a feedforward neural network can emulate any function, but the neural network cannot replicate the operation of any computer program.

Just as for other statistical models, it is often necessary to explain why a neural network produced the output that it did. One means of explaining a neural network is to use an algorithm to rank the importance of each of the network's inputs. Garson (1991) created this type of algorithm that could rank the importance of the input neurons by partitioning hidden to output connection weights into components associated with each input neuron using absolute values of connection weights. Olden and Jackson (2002) achieved greater accuracy than the Garson algorithm by introducing the connection weights algorithm that used only the product of the input to hidden and hidden to output connection weights between each input neuron and output neuron and sums the products across all hidden neurons. The Garson and connection weights algorithms are both dependent on the model being a single-hidden layer neural network. In fact, many models have feature-ranking algorithms that are specific to that model. These ranking algorithms are called model dependent because they work only for a single type of model. Because the weight connections and Garson algorithms assume a single layer neural network, they are not compatible with deep learning.

Breiman (2001) introduced a feature importance algorithm in his seminal paper on random forests. Although he presented this algorithm in conjunction with random forests, it is model-independent and appropriate for any supervised learning model. This algorithm, known as the input perturbation algorithm, works by evaluating a trained model's accuracy with each of the inputs individually shuffled from a data set. Shuffling an input causes it to become useless—effectively removing it from the model. More important inputs will produce a less accurate score when they are removed by shuffling them. This process makes sense, because important features will contribute to the accuracy of the model.

Several feature ranking algorithms, including the Garson and input perturbation, were empirically compared by Olden, Joy, and Death (2004). Input perturbation was found to be among the most accurate feature ranking algorithms surveyed. The best algorithm found by their research was the weight connections algorithm. However, because the weight connections algorithm is incompatible with deep learning, it was not useful to this dissertation research. Because the input perturbation algorithm is compatible with any model, including deep neural networks, it played a prominent role in this dissertation. The algorithm developed by this dissertation used an objective function partly based on the ranking of the features engineered by a genetic programming algorithm.

By definition, a feedforward neural network contains only forward connections. Thus, the input layer connects only to the first hidden layer, the first hidden layer connects only to the second hidden layer, and the final hidden layer connects only to the output layer. However, recurrent neural networks allow connections to previous layers. Elman (1990) and Jordan (1997) began the research of recurrent neural networks and introduced their simple recurrent networks (SRNs) as the Elman and Jordan neural networks. Figure 3 shows an Elman neural network while Figure 4 shows a Jordan neural network.

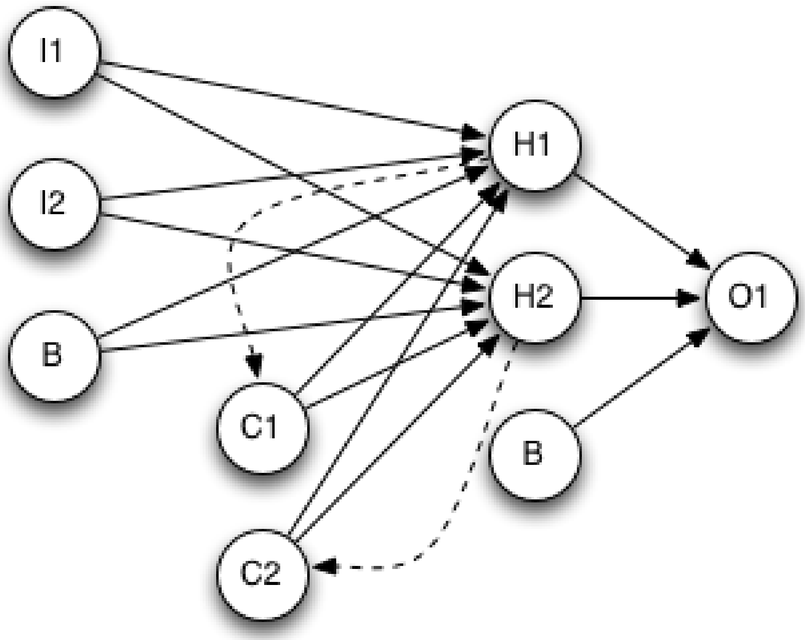


Figure 3. Elman neural network

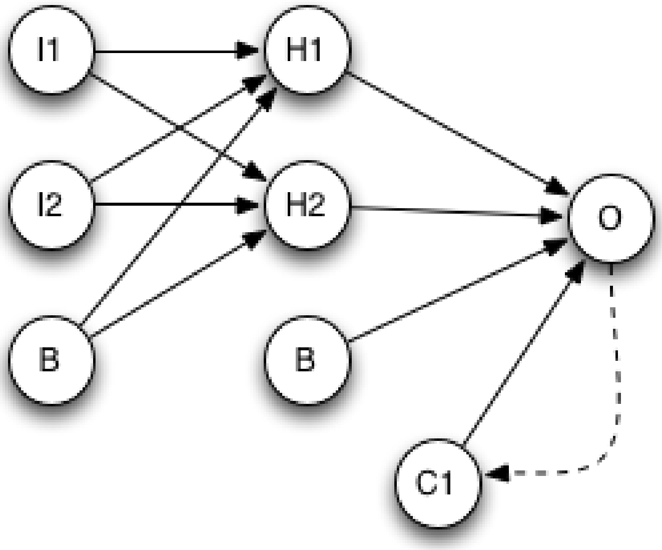


Figure 4. Jordan neural network

Both the Elman and Jordan networks contain a context layer (C) that will initially output 0. However, the context layer always remembers the input from the previous time that the network was calculated. Subsequent calculations of the neural network will cause the context layer to always output the input received by the layer in the previous iteration. The data received are stored and then output at the next calculation of the neural network. The context layer in a SRN can function somewhat like a time-loop in that the output is always the input from the previous iteration.

A technique known as backpropagation through time can usually train recurrent neural networks (Mozer, 1989; Robinson & Fallside, 1987; Werbos, 1988). This technique is like backpropagation except that it unfolds the recurrent layers to make the recurrent neural network appear as one large feedforward network. Backpropagation through time has a configuration parameter that specifies the number of time slices that the program can unfold into the network. Several virtual layers equal to that configuration parameter can create the virtual network. The same backpropagation algorithm that was used for feedforward networks trains the virtual network.

Feedforward neural networks will always produce the same output for a given feature vector. However, recurrent neural networks will maintain state from previous computations. This state will affect the neural network output; therefore, the order that feature vectors are presented to the neural network will affect the output. This capability makes recurrent neural networks applicable to time series prediction. For recurrent neural networks, a series of events, represented as individual feature vectors, now produces an output. This result differs from a single feature vector in regular feedforward neural networks. While recurrent neural networks are particularly adept at handling time-series, there are other alternatives, Balkin and Ord (2000) demonstrated encoding methods to use regular non-recurrent feedforward neural networks with time series prediction.

One issue with SRN networks, such as Elman and Jordan, is that the longer a time series becomes, the less relevant a context layer. To overcome this problem, Hochreiter and Schmidhuber (1997) introduced the long short-term memory (LSTM) network, which is shown in Figure 5.

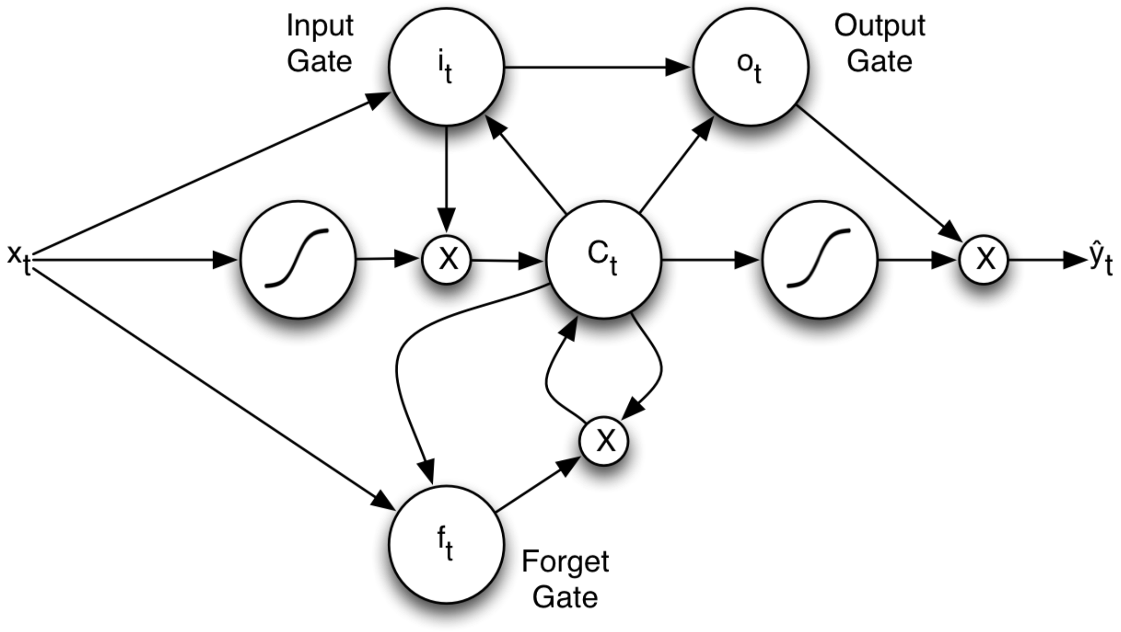


Figure 5. Long short-term memory (LSTM)

It is important to note that this figure shows only a single neuron of a LSTM network. These LSTM neurons can be placed inside of regular feedforward neural networks. Usually, LSTM neurons are placed as an entire layer of such neurons. On a conceptual level, the LSTM neuron functions similarly to the context neurons of the SRNs. The C-labeled node, near the center of the figure, represents the context memory of the node. However, unlike SRN's, the LSTM does not simply copy the previous neural network's computation to its internal state. The LSTM uses three gates to control when the input is accepted, when the internal state is forgotten, and when the internal state is fed to the next unit. These gates are activated when the input reaches a threshold specified by a trained internal weight. The backpropagation through time algorithm trains the gate threshold weights along with every other weight. Because it controls the internal state, input, and output, the LSTM is considerably more effective at recalling time series than a regular SRN.

The inability to learn large numbers of hidden layers was not the only barrier to widespread neural network adoption. One problem that remains for neural networks is the large number of hyper-parameters that a neural network contains. A neural network practitioner must decide the number of layers for the network as well as the quantity of neurons that each of the hidden layers will contain. Prior research in the field of neural networks reveals that researchers have long aspired to create an algorithm that automatically determines the optimal structure for neural networks. Stanley and Miikkulainen (2002) invented the NeuroEvolution of Augmenting Topologies (NEAT) neural network that utilizes a genetic algorithm to optimize the neural network structure. The genetic algorithm searches for the best neural network structure and weight values to minimize the loss function.

Although most neural network research has shifted towards deep learning, some research remains on classical neural network structures. Chea, Grenouillet, and Lek (2016) used a self-organizing map (SOM) to predict water quality. A SOM allowed M. Wang et al. (2016) to identify the mixtures of Chinese herbal medicines. Sobkowicz

(2016) implemented a NEAT neural network to perform sentiment analysis in the Polish language.

Deep Learning

While it is theoretically possible for a single-hidden layer neural network to learn any problem (Hornik, 1991), this outcome will not necessarily occur in practice. Additional hidden layers can allow the neural networks to learn hierarchies of features, thereby simplifying the search space. They can also find the optimal set of weights with less training. Unfortunately, no method to train these networks existed until the development of a series of innovations that introduced new transfer functions and training methods. McCulloch and Pitts (1943) introduced the seminal neural network calculation, shown in Equation 1. Many new technologies have built upon this core calculation.

Hinton et al. (2006) first implemented a deep neural network with favorable results. They created a learning algorithm that could train deep network variants like those that Fukushima (1980) introduced for belief networks. This discovery renewed interest in deep neural networks. Several additional technologies, such as stochastic gradient descent (SGD) (Bertsekas, 1999), rectified linear units (ReLU) (Glorot, Bordes, & Bengio, 2011), and Nesterov momentum (Sutskever, Martens, Dahl, & Hinton, 2013), have made training of deep neural networks more efficient. Taken together, these and other technologies are referred to as deep learning.

Neural networks with many layers will often experience a problem in which the gradients become 0 with certain transfer functions. Hochreiter (1991) was the first to describe this vanishing gradient problem in his Ph.D. thesis. Prior to deep learning, most neural networks used a simple quadratic error function on the output layer (Bishop, 1995). De Boer, Kroese, Mannor, and Rubinstein (2005) introduced the cross-entropy error function, and it often achieves better results than the simple quadratic because it addresses the vanishing gradient problem by allowing errors to change weights even when a neuron's gradient saturates (their derivatives are close to 0). It also provides a more granular means of error representation than the quadratic error function for classification neural networks.

Weight Initialization

Neural networks must start with random weights (Bengio, 2009). These random weights are frequently sampled within a specific range, such as (-1,1). However, this simple range initialization can occasionally produce a set of weights that are difficult for backpropagation to train. As a result, researchers have shown interest for weight initialization algorithms that provide a good set of starting weights for backpropagation (Nguyen & Widrow, 1990). Glorot and Bengio (2010) introduced what has become one of the most popular methods called the Xavier weight initialization algorithm. Because of its ability to produce consistently performing weights suitable for backpropagation training, the research in this dissertation will use the Xavier weight initialization algorithm.

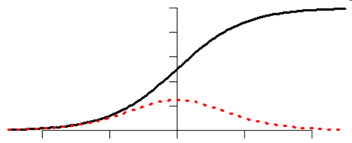
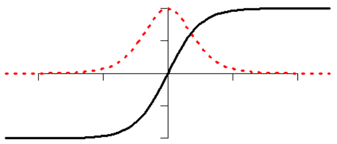
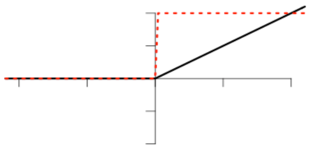
Backpropagation relies on the derivatives of the transfer functions to incrementally calculate, or propagate, error corrections from the output neurons back through the weights of a neural network. Prior to 2011, most neural network hidden layers used the hyperbolic tangent or the logistic transfer function, which are sigmoidal transfer functions. The derivative of both of these functions saturate to 0 as *x* approaches either positive or negative infinity, causing these transfer functions to exhibit the vanishing gradient problem. Glorot et al. (2011) introduced the rectified linear unit (ReLU) transfer function to address this problem.

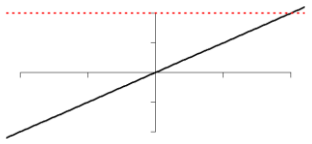
Transfer Functions for Deep Learning

The ReLU transfer function usually achieves better training results for deep neural networks than the sigmoidal transfer functions. According to current research (Bastien et al., 2012), the type of transfer function to use for deep neural networks is well defined for each layer type. For their hidden layers, deep neural networks employ the ReLU transfer function. For their output layer, most deep neural networks utilize a linear transfer function for regression, and a softmax transfer function for classification. No transfer function is needed for the input layer. Bastien et al. (2012) present research that follows this form and uses the ReLU transfer function for hidden layers and either linear or softmax for the output layer.

Table *1* summarizes the logistic, hyperbolic tangent, ReLU, linear, and softmax transfer functions:

Table 1. Common neural network transfer functions

Name/ Range	Expression (Forward)	Derivative (Backward)	Graph (Derivatives in Red)
Logistic/ Sigmoid [0,1]	$\phi(x) = \frac{1}{1+e^{-x}}$	$\phi'(x) = \frac{e^x}{(1+e^x)^2}$	
HTan [-1,1]	$\phi(x) = \text{htan}(x)$	$\phi'(x) = 1 - \phi^2(x)$	
ReLU [0,+∞)	$\phi(x) = \max(0, x)$	$\phi'(x) = \begin{cases} x > 0 & 1 \\ x \leq 0 & 0 \end{cases}$	

Linear (-∞,+∞)	$\phi(x) = x$	$\phi'(x) = 1$	
Softmax (-∞,+∞)	$\phi(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{\lVert x \rVert} e^{x_k}}$	NA	NA

This table indicates why ReLU often achieves better performance than a logistic or hyperbolic tangent transfer function. The graph column of the table shows both the transfer function as well as the derivative of that transfer function. The solid black line refers to the transfer function output, and the dotted red line is the derivative. Both sigmoid-shaped transfer functions have derivatives that quickly saturate to 0 as it approaches either positive or negative infinity. The ReLU, on the other hand, does not saturate as positive infinity is approached. Additionally, the much wider range of the ReLU lessens the need for the common practice of normalizing the inputs to values closer to the range of the sigmoidal transfer function in use.

Regularization

Overfitting is a frequent problem for neural networks (Masters, 1993). A neural network is said to be overfit when it has been trained to the point that the network begins to learn the outliers in the data set. This neural network is learning to memorize, not generalize (Russell & Norvig, 1995). A class of algorithms designed to combat overfitting is called regularization algorithms. One of the most common forms of neural network regularization is to simply add a scaled summation of the weights of the neural network to the loss function. This calculation will cause the training algorithm to attempt to lower the weights of the neural network along with the output error. Two of the most common forms of this weight regularization are L1 and L2 (Ng, 2004).

L1 regularization, shown in the following equation, sums the weights of the neural network (*w*) and produces an error value (*E₁*) that is added to the loss function of the neural network.

$$\boldsymbol{E_1 = \frac{\lambda_1}{n} \sum_{w} |w|} \tag{6}$$

It is important to note that the *w* vector includes only actual weights and not bias-weights. The value λ_1 is a scaling factor for the effect of the L1 regularization. If λ_1 is too high, the objective of lowering the weights will overwhelm the one for achieving a lower error for the neural network training. This situation causes a failure of the neural network to converge to a low error. The value *n* represents the number of training set elements. L2 regularization is defined similarly to L1 and is provided by the following equation:

$$\boldsymbol{E_2 = \frac{\lambda_2}{n} \sum_{w} w^2} \tag{7}$$

Both L1 and L2 regularization sum the weights without regard to their sign. This magnitude-oriented approach is accomplished by an absolute value for L1 and a square for L2. The weights are pushed towards 0 in both cases. However, L1 has a greater likelihood of pushing the weights entirely to 0 and effectively pruning the weighted connection (Ng, 2004). This pruning feature of L1 is especially valuable for the dissertation research because it can function as a type of feature selection. L1 will indicate worthless engineered features by pruning them.

L1 and L2 are not the only forms of regularization. Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014) introduced dropout as a simple regularization technique for deep neural networks. Dropout is typically implemented as a single dropout layer, as demonstrated by Figure 6:

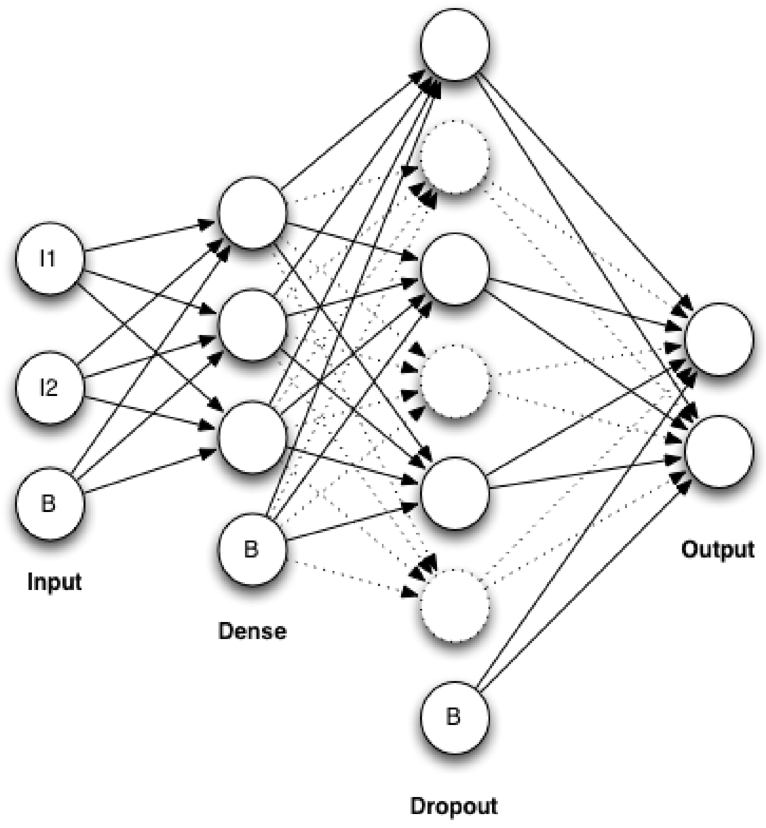


Figure 6. Dropout layer in a neural network

The dashed lines in Figure 6 represent the dropped neurons. In each training iteration, some neurons are removed from the dropout layer. However, neither the bias neurons nor the input and output neurons are ever removed. When a neuron is discarded, the training iteration occurs as if that neuron and all its connections are not present. However, the drop is only temporary; the neuron and its connections will return in the next iteration, and a different set is removed. In this way, dropout decreases overfitting by preventing the network from becoming too dependent on any set of neurons. Once training is complete, all the neurons return. Dropout affects only neural networks during the training phase.

Enhancements to Backpropagation

There have been several important enhancements to the basic backpropagation weight update rule. Momentum has been a significant component of backpropagation training for some time. Polyak (1964) introduced the seminal momentum algorithm that is a regularization technique for gradient ascent/descent. Momentum backpropagation adds a portion of the previous iteration's weight change to the current iteration's weight change.

$$\boldsymbol{v_t = \gamma v_t - \eta \nabla_{\theta_{t-1}} J(\theta_{t-1})} \tag{8}$$

Consequently, the weight updates have the necessary momentum to push through local minima and to continue the descent of the output of the loss function.

Nesterov momentum (Nesterov, 1983) further enhances the momentum calculation and increases the effectiveness of random mini-batches selected by SGD. Nesterov momentum decreases the likelihood of a particularly bad mini-batch from changing the weights into an irreparable state. A neural network update rule using Nesterov momentum was introduced by Sutskever et al. (2013).

Researchers have developed several innovations beyond the classic backpropagation and Nesterov momentum update rules. Classic backpropagation, even with Nesterov momentum requires that researchers choose learning rate and momentum training parameters that are applied across all weights in the neural network. It is usually advantageous to decay the learning rate as the neural network trains (Bottou, 2012).

Additionally, each weight in a neural network might benefit from a different learning rate (Riedmiller & Braun, 1992). Duchi, Hazan, and Singer (2011) introduced the Adaptive Gradient algorithm (AdaGrad) to both decay the learning rate, as well as vary this rate per weight. Zeiler (2012) attempted to mitigate the aggressive monotonic learning rate decay of AdaGrad by introducing the Adadelta update rule that only uses a window of gradients that determine the learning rate. Tieleman and Hinton (2012) independently proposed RMSprop to address the aggressive learning rate decay of AdaGrad. The algorithm determines the learning rate by dividing gradients by a root mean square of the weights.

Kingma and Ba (2014) introduced the Adam update rule that derives its name from the adaptive moment estimates that it uses. Adam estimates the first (mean) and second (variance) moments to determine the weight corrections. Adam begins with exponentially decaying averages of past gradients (*m*):

$$\boldsymbol{m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t} \tag{9}$$

This average accomplishes a similar goal as that of classic momentum update; however, its value is calculated automatically based on the current gradient (g_t). The update rule then calculates the second moment (v_t):

$$\boldsymbol{v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2} \tag{10}$$

The values m_t and v_t are estimates of the first moment (the mean) and the second moment (the un-centered variance) of the gradients. However, they will have a strong bias towards zero in the initial training cycles. The first moment's bias is corrected as follows:

$$\widehat{\boldsymbol{m}_t} = \frac{\boldsymbol{m}_t}{\mathbf{1} - \boldsymbol{\beta}_1^t} \tag{11}$$

Similarly, the second moment is also corrected:

$$\widehat{\boldsymbol{v}_t} = \frac{\boldsymbol{v}_t}{\mathbf{1} - \boldsymbol{\beta}_2^t} \tag{12}$$

These bias-corrected first and second moment estimates are applied to the ultimate Adam update rule, as follows:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \frac{\boldsymbol{\eta}}{\sqrt{\widehat{\boldsymbol{v}_t}} + \boldsymbol{\epsilon}} \widehat{\boldsymbol{m}_t} \tag{13}$$

This dissertation featured the Adam update rule for all neural network training due to the rule's robustness to initial learning rate (η) and other training parameters. Kingma and Ba (2014) propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ε, that were used in this dissertation.

Researchers have also utilized deep learning for recurrent neural networks. The research community has recently shown considerable interest in deep LSTM networks. Kalchbrenner, Danihelka, and Graves (2015) used a grid of LSTM units to achieve greater accuracy. Chung, Gulcehre, Cho, and Bengio (2015) introduced the gated recurrent network (GRU) and added an output gate, which allows greater accuracy as the time series increases in length. Unlike feedforward neural networks, LSTM and GRU are recurrent networks that can function as Turing machines (Graves et al., 2014).

Evolutionary Programming

Holland (1975) introduced evolutionary algorithms. Later, Deb (2001) extended this work to present genetic algorithms as they are known today. The genetic algorithm is a generic population-based metaheuristic technique to find solutions to many real-world search and optimization problems. Darwinian evolution inspired these algorithms. A population of potential solutions is evolved as the fittest population members produce subsequent generations through the genetic operators of crossover and mutation. Each of these potential solutions is referred to as either a genome or a chromosome (depending on the implementation). This evolutionary process is a search for the classic balance between exploitation and exploration. Mutation and crossover provide the genetic algorithm with the ability to explore and exploit the search space. Exploration occurs when the mutation genetic operator introduces randomness to the population. The crossover genetic operator exploits by creating new members containing traits from the best members of the population (Holland, 1975).

The genetic algorithm represents potential solutions as fixed-length vectors. This vector might be the weights of a neural network, coefficients of an expression, or any other fixed-length vector that must be optimized against an objective function. Mutation occurs by randomly perturbing the elements of a vector. Crossover is achieved by splicing together the vectors of two or more fit parent vectors.

An objective function serves to evaluate the population. The loss function of a neural network is somewhat like the evolutionary algorithm's objective function. Both the loss function and objective function provide a numeric value to be optimized. Some evolutionary algorithms also allow the objective function to be maximized. The choice between minimization and maximization depends on the domain of the problem. Although the loss function and objective function both accomplish similar goals, it is standard procedure to refer to the evaluation function for an evolutionary algorithm as an objective function.

While many problems can be modeled as a fixed-length vector, this representation is a limiting factor for classic genetic algorithms. They will never improve the underlying neural network algorithm even though they evolve the weights to produce better results. To evolve better algorithms, the computer programs themselves must become the genomes that are evolved (Poli, Langdon, & McPhee, 2008). Genetic programming is an answer to the fixed-length limitation of classic genetic algorithms.

Rather than evolving a fixed-length vector, genetic programming evolves representations of actual computer programs to achieve an optimal score to an objective function. Koza (1992) popularized this active area of research to automatically generate programs to solve specific problems. Much of Koza's research represents the genetic programs as trees. Although most genetic programming research has been focused on tree representation (White et al., 2013), there are other representations of genetic programs, such as grids and probabilistic structures (Banzhaf, Francone, Keller, & Nordin, 1998). This dissertation research used only the tree representation of genetic programs.

A tree-based genetic program is implemented as a directed acyclic graph (DAG). The tree is composed of connected nodes. The tree starts with a parentless root node. It connects to other nodes and points to still more nodes. Interior nodes with at least one child form the tree. Terminal nodes without children also exist. Ultimately, the tree reaches all the terminal nodes, which represents the variables and constants. In turn, the operators using the variables and constants are composed of the interior nodes. The following expression could be represented as a tree for genetic programming:

$$\frac{\boldsymbol{x}}{\mathbf{2}} - \mathbf{1} + \mathbf{3cos}(\boldsymbol{y}) \qquad (14)$$

It is common in computer science to represent these expressions as trees. Lisp and Scheme are early programming languages that utilized similar tree representations called S-Expressions (Sussman, Abelson, & Sussman, 1983). Figure 7 shows the expression mentioned above as a tree:

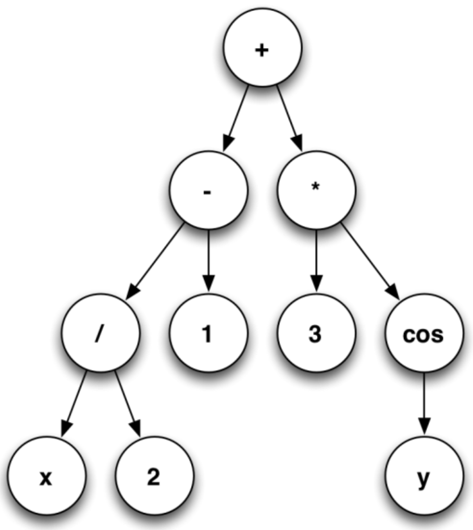


Figure 7. Expression tree for genetic programming

It is also possible to express entire computer programs as trees. The branching nature of a tree can encode if-statements and loops. The programs encoded into these types of trees can be Turing complete (Teller, 1994), which means they can theoretically compute anything (Turing, 1936). Additionally, nodes can be created to change the values of variables. However, trees are not the only representation for genetic programs.

Much of the research into genetic programming has involved the best way to represent the genetic programs (Poli et al., 2008).

All evolutionary algorithms must have processes for exploitation and exploration. For genetic programming, crossover and mutation can accomplish these functions. The representation of the underlying data dictates the exact nature of the mutation and crossover algorithms. Koza (1992) defined several possible algorithms for crossover and mutation of genetic programming trees. The two most popular are point crossover and subtree mutation. This dissertation includes point crossover and subtree mutation for its genetic programming solution. Figure 8 shows point crossover:

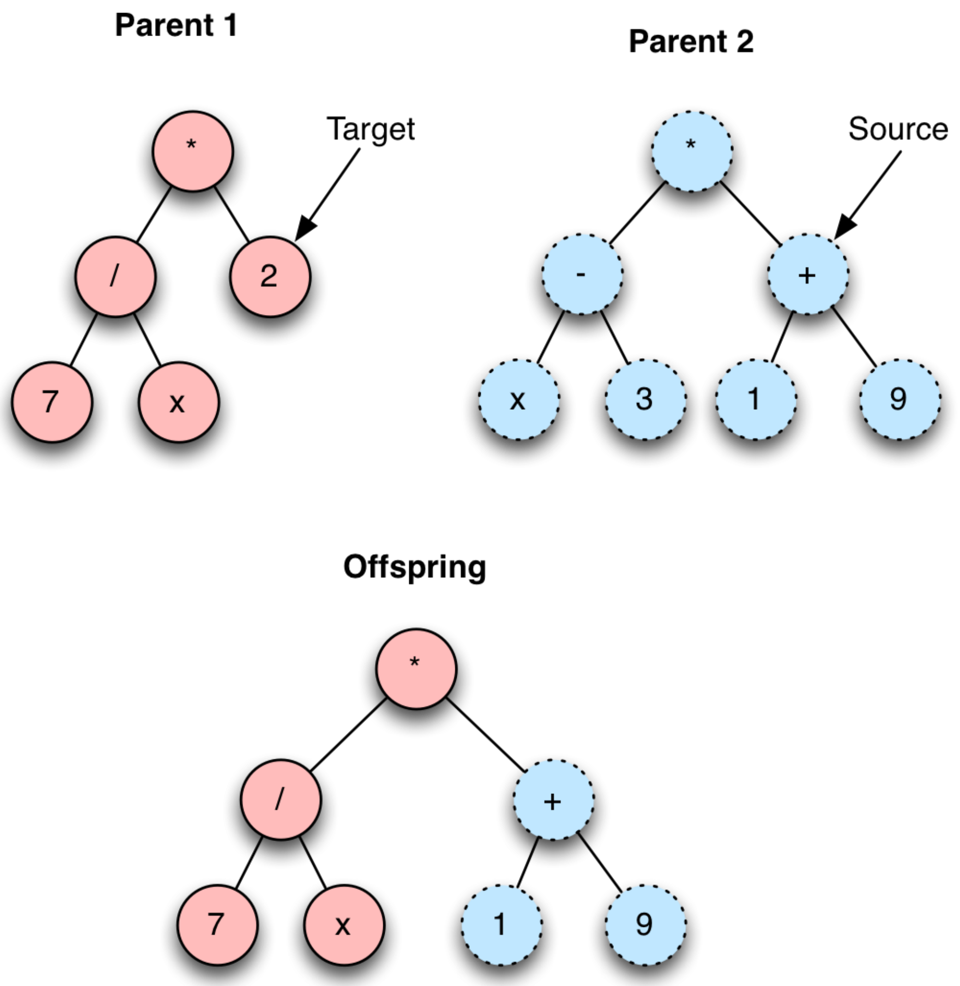


Figure 8. Point crossover

To implement point crossover, two parents are chosen. The algorithm chooses a random target node for parent 1 and a random source node for parent 2. Then it creates the offspring by cloning parent 1 and replacing the target node on the offspring with a copy of the subtree at the source node on parent 2. The crossover operation does not modify either parent, and the offspring may be more fit than the parents. Most importantly, crossover does not introduce new information; it only recombines existing information. Thus, crossover exploits rather than explores.

Koza also defined subtree mutation, as shown in Figure 9:

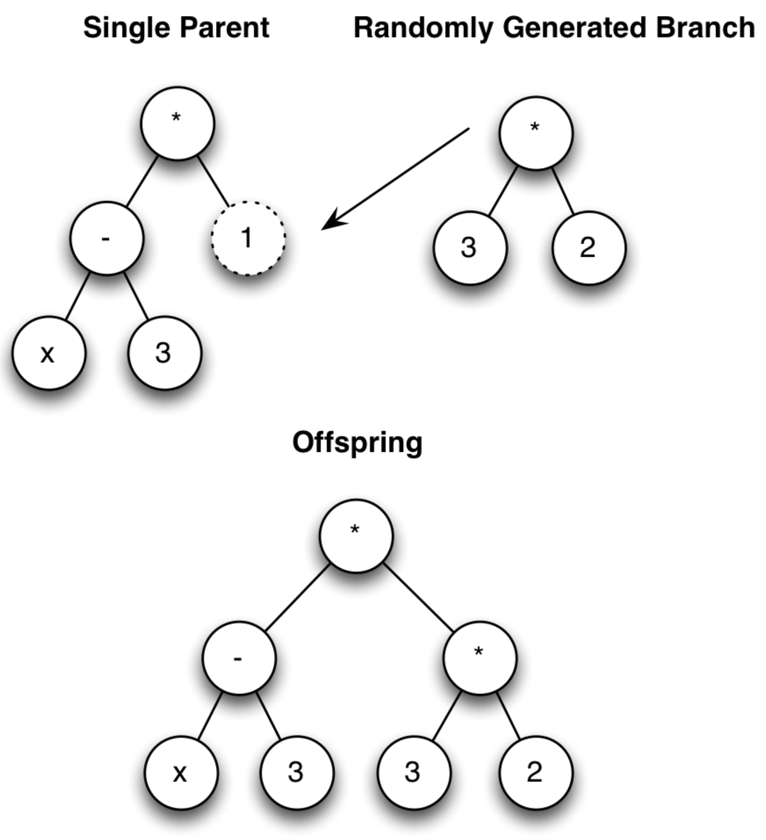


Figure 9. Subtree mutation

Mutation uses only a single parent, and, like crossover, the operation does not modify the parent. The algorithm chooses a random insertion node on the parent and generates a new random branch. Cloning the single parent and grafting the randomly generated branch onto the offspring at the previously chosen random insertion node creates the offspring. It is important to note that the mutation operator does add new information to the genome and, therefore, explores rather than exploits.

Speciation

It can be difficult to produce viable offspring from a crossover operation between two dissimilar genomes. This fact is true in real life as well. Two animals are considered to be from the same species if they are capable of producing viable offspring (Dawkins, 1976). In genetic programming, if the two parent genomes are similar, the probability increases for the crossover operator to produce offspring superior to the parents (Poli et al., 2008). Although there are several speciation strategies, this dissertation uses the strategy included with the Encog framework that is similar to the speciation algorithm of Stanley and Miikkulainen (2002) for the NEAT neural network. In that algorithm, the larger of the two parent trees is chosen, and the percentage of the same nodes of the larger tree that are also in the smaller tree is calculated. Genomes that have a percent similarity above a configurable threshold are in the same species. The species divisions are recalculated at the end of the training iterations. Thus, a child is not necessarily in the same species as the parents.

Other Genetic Program Representations

Trees are not the only means of representing genetic programs. Modern computer hardware portray computer programs as a series of linear instructions (Knuth, 1997), not as trees. Although these programs can be written as trees, it is often not practical because the linear nature of programming will create tall, unbalanced trees. This problem led a number of researchers to investigate a linear representation of the genetic programs. Poli et al. (2008), Banzhaf (1993), Perkis (1994), and Diplock (1998) sought to implement genetic programing in a fashion that mirrored the linear computer architecture. Nordin (1994), Nordin, Banzhaf, and Francone (1999), Crepeau (1995), and Miller and

Thomson (2000) went even further and evolved bit patterns that represented the actual CPU machine language instruction codes. The linear genetic programming systems create code that closely resembles pseudocode. Thus, a human programmer can more easily interpret linear genetic programs than a tree-based genetic program.

Cartesian Genetic Programming (CGP) (Miller & Thomson, 2000) represents the evolvable genetic programs as two-dimensional grids of nodes. CGP easily encodes computer programs, electronic circuits, neural networks, mathematical expressions, and other computational structures. Fixed-length vectors serve as integer-based grids for the crossover and mutation genetic operators.

Many genetic programs operate exclusively on floating point data. While some research into genetic programming have introduced Booleans, strings and other types, often only a single type is used. Such types are important; as traditional computer programs frequently use many different data types. If genetic programming is to evolve actual computer programs, it is important to offer this same multi-type flexibility. Unfortunately, data types such as *integer*, *string*, and *structure* complicate genetic programming because not every operator can accept all types. For example, the subtraction operator (“-”) can easily be applied to integers and floating-point numbers, but it is generally undefined for strings or Booleans. To overcome this limitation, Worm and Chiu (2013) created a system of grammar rules that document dependencies between operators and data. For example, the subtraction operator (“-”) works with numbers but not strings. These rules restrict the crossover and mutation operators and ensure that new programs are valid.

Genetic Programming for AFE

Examples of the application of genetic programming to feature engineering exist in previous machine learning literature. Xue, Zhang, Browne, and Yao (2016) provide a very thorough and recent survey of evolutionary computation application to AFE in literature. They divide genetic programming based AFE algorithms into two categories: filter and wrapper approaches. A wrapper approach evaluates features produced by the AFE with an actual model type, such as a neural network. A filter approach scores individual genomes with a proxy algorithm. Because neural networks, and other models, can be computationally expensive, filter approaches provide a means of evaluating search spaces that would be too large for a wrapper approach. The AFE developed by this dissertation is a wrapper approach.

Xue et al. (2016) further classified wrapper genetic programming based AFEs as being either single-objective or multi-objective. Single-objective simply search for any combination of features that improve accuracy. Their paper did not find any prior research of multi-objective wrapper genetic programming based AFE algorithms. The AFE provided by dissertation is multi-objective because it seeks engineered features that neural networks cannot easily synthesize on their own. Because of this, the AFE developed by this dissertation is a novel approach. Xue et al. (2016) found 31 other papers that made use of wrapper based genetic programming. However, unlike the algorithm presented in this dissertation, none were specifically designed for deep neural networks.

Although these examples are similar to the AFE developed by this dissertation, the method in this dissertation is unique. Specifically, this dissertation feature the leveraging of unique characteristics of deep neural networks to constrain the large search space of potential engineered features. Additionally, the algorithm in this dissertation introduces a novel objective function for genome selection and evaluates many potential engineered feature genomes simultaneously.

Guo et al. (2005) demonstrated that feature engineering could take advantage of genetic programming within the domain of fault classification. These researchers employed the Fisher criterion as the fitness function for the genetic program. This criterion is typically used in conjunction with Linear Discriminant Analysis (LDA) (Fisher, 1936). The criterion also measures the inter-class scatter between two classes in a classification problem. The work of Guo et al. experimented with expressions of original features to linearly separate pairs of classification outcomes in the training set. This approach differs from this dissertation research because it does not attempt to exclude engineered features that the neural network can easily synthesize on its own. In other words, the research of Guo et al. is not specifically tailored to deep neural networks. Guo et al. evaluated the results of feature engineering using a shallow 14-neuron network and a support vector machine.

Additionally, Neshatian (2010) provided another application of genetic programming to feature engineering by introducing a genetic programming-based feature engineering algorithm. He demonstrated that objective functions that utilize an underlying predictive algorithm are computationally expensive. Rather than directly targeting one specific predictive model, the Neshatian algorithm uses a *decision stump,* which is essentially a single node on a decision tree. This single stump learns a threshold value that separates two classes with minimal overlap. In this way, the approaches of both Guo et al. (2005) and Neshatian (2010) are similar. To see the combined approach, consider a linear separation between two iris species in the classic Iris data set (Fisher, 1936), shown in Figure 10.

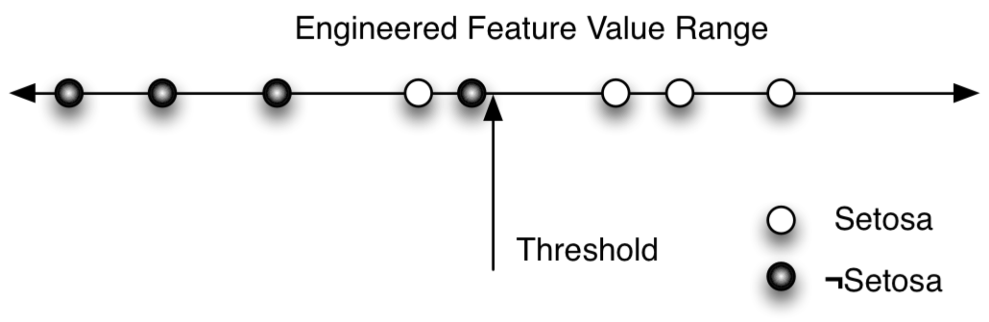


Figure 10. Feature engineering to linearly separate two classes

As observed, the genetic programming algorithm is seeking an engineered feature expression that isolates two of the classes (in this case iris species Setosa and the other two species). The commonly used iris data set contains the species and four measurements of 150 iris flowers of three species. The engineered feature becomes what is commonly referred to as a one-versus-rest model to distinguish one classification from the rest. Both Neshatian (2010) and Guo et al. (2005) demonstrated this model to benefit feature engineering for decision trees and shallow neural networks. However, Neshatian (2010) noted the inability of the algorithm to engineer features that enhanced neural network prediction, attributing this failing to the fact that neural networks synthesize comparable engineered features on their own. However, deep neural networks do benefit from feature engineering (Bengio, 2013; Blei et al., 2003; M. Brown & Lowe, 2003; Coates et al., 2011; Coates & Ng, 2012). Prior research also empirically demonstrated the benefits of feature engineering to deep neural networks (Heaton, 2016). Thus, the main purpose of this dissertation was to find these engineered features that increased the accuracy of deep neural networks.

Summary

Genetic programming and deep neural networks both accomplish similar tasks. They accept an input vector and produce either a numeric or categorical output. However, internally, they function differently. Deep learning adjusts large matrices of weights to produce the desired output. Genetic programming constructs computer programs to yield the desired output. A genetic program can be Turing complete; however, a feedforward neural network is not generally Turing complete.

Feature engineering is a technique that preprocesses the data set to transform it into a new data set designed to better fit a model and to achieve better accuracy. Typically, feature engineering is a manual process. However, research interest in automating aspects of feature engineering exists. Because genetic programming is capable of evolving programs and expressions from its input features, it is logical to make it the basis for a feature engineering algorithm. Creating this algorithm was the goal of this dissertation. The exact methodology that was used to build this algorithm is discussed in the next chapter.

Chapter 3

Methodology

Introduction

For this dissertation research, an algorithm was built that was capable of the automated creation of engineered features that could increase the accuracy of a deep neural network. This algorithm was designed to be a metaheuristic search over all combinations of the original feature set and had the goal to return features from this search space that had the potential of increasing the accuracy of a deep neural network. Because the search space is mathematical expressions, genetic programming was a natural choice of a metaheuristic.

This chapter includes the description of five experiments that provided background information for the design of this AFE algorithm. These experiments provided insights into search space constraints, objective function design, and other areas important to the creation of a genetic programming solution. The sixth experiment described at the end of the chapter provided a means for benchmarking the AFE algorithm.

Algorithm Contract and Specification

For the purposes of this research, a deep neural network composed of the following standard components was targeted:

- Layered Feedforward Neural Network (Rumelhart et al., 1985).
- ReLU or Sigmoid Transfer Function for Hidden Layers (Glorot et al., 2011).
- Xavier Weight Initialization (Glorot & Bengio, 2010).
- Adam Update Rule (Kingma & Ba, 2014) with Stochastic Gradient Descent (Bottou, 2012).

For a complete listing and description of the algorithms used for the AFE algorithm, refer to Appendix A, “Algorithms.” The neural network algorithms were not modified to achieve the goal of this research. The feature engineering algorithm produced by this research is compatible with a standard feedforward deep neural network that was supported by any common deep learning framework, such as Theano (Bastien et al., 2012; Bergstra et al., 2010), CNTK (D. Yu et al., 2014), TensorFlow (Abadi et al., 2016) or Encog (Heaton, 2015).

It was important to use a standard deep feedforward neural network because the algorithm would have limited application if its engineered features could only increase accuracy of non-standard implementations of deep learning. The accuracy achieved by different deep learning frameworks, such as Encog, Theano or TensorFlow will likely be different, due to implementation details in these frameworks. Encog was the only deep learning framework in the scope of this dissertation.

The algorithm developed accepts an operating specification encoded as a YAML file, which the following example shows:

```
input_dataset: input.csv
input_x: [ 'acceleration', 'horsepower', 'cylinders' ]
input_y: [ 'mpg' ]
input_headers: true
output_features: features.csv
output_augmented_dataset: augmented_input.csv
prediction_type: regression
transfer_hidden: relu
...other configuration settings...
```

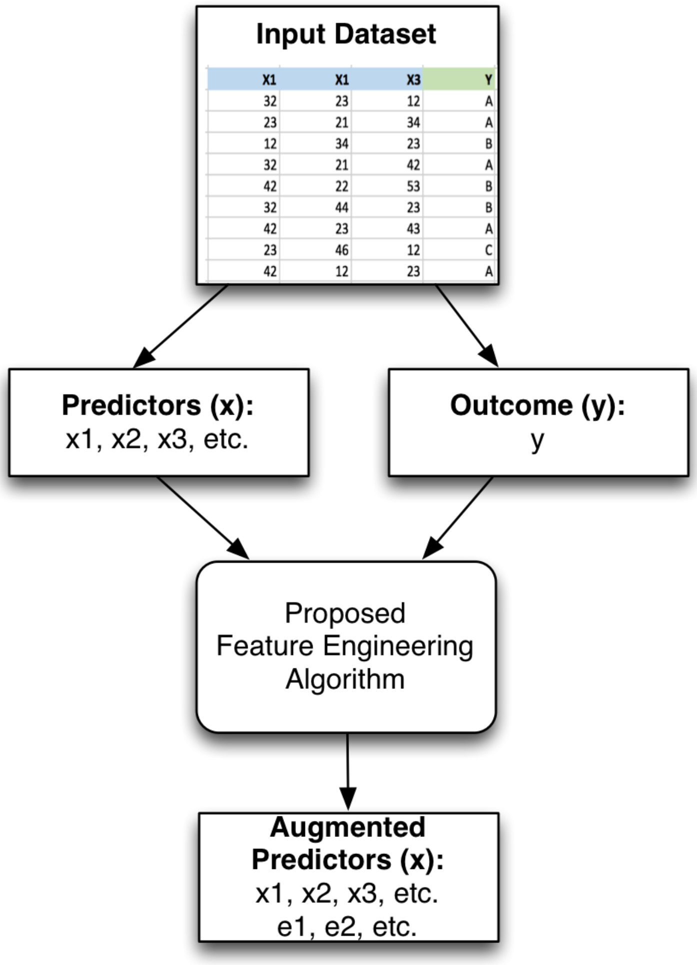
The above file specified the configuration to analyze the UCI AutoMPG data set. The features *acceleration*, *horsepower* and *cylinders* allowed the model to make a prediction. In this case, the prediction was *mpg*. The features and prediction were all column names in the original file.

The configuration items have the following purposes:

- **input_dataset:** The name of the data set that contained the predictors (*x*) and expected outcomes (*y*). The algorithm would use only the specified columns, except the target column(s) as the input.
- **input_x:** The list of column names used as predictors (*x*). The names could be either symbolic or the zero-based index of the column.
- **input_y:** The list of column names used as the target(s) (*y*). The names could be either symbolic or the zero-based index of the column.
- **input_headers:** Boolean that indicated if the input CSV had headers. If there were no headers, the columns were referenced by their zero-based index.
- **output_features:** A CSV file that contained a summary of the engineered features.

- **output_augmented_dataset:** The output CSV file that contained the input CSV file data, along with the new engineered features.
- **prediction_type:** The type of prediction and was either classification or regression.
- **transfer_hidden:** The type of hidden transfer function targeted. It was either *relu*, *sigmoid* or *htan*.

The algorithm's input file was the same input file that trained a neural network. The output file appended engineered feature columns. This output file trained a neural network. All common neural network frameworks could train from files of this format. The goal of this research was to show that a neural network trained from the algorithm's augmented input would produce a neural network that would allow a more accurate prediction for some data sets. Figure 11 summarizes this high-level design:



X1	X1	X3	Y
32	23	12	A
23	21	34	A
12	34	23	B
32	21	42	A
42	22	53	B
32	44	23	B
42	23	43	A
23	46	12	C
42	12	23	A

Figure 11. Algorithm high level design and contract

Other than z-score standardization, the AFE algorithm did not perform any preprocessing on the input data set. All values in this data set were assumed to be numeric and any missing values properly handled beforehand. For information on how data sets were preprocessed for this dissertation, refer to Appendix D, “Data Set Descriptions.”

Algorithm Design Scope

The AFE algorithm utilized the genetic programming components of Encog, which implemented a genetic programming framework according to Koza (1992). The software developed centered primarily on the following aspects of genetic programming:

- Constraints to narrow the search domain.
- Objective function to effectively evaluate candidate features.
- Post processing analysis of genetic population.

The following sections highlight the approach for each of these components:

Narrowing the Search Domain

Placing search constraints upon a genetic programming algorithm is possible (Gruau, 1996; Janikow, 1996). These limits are sometimes implemented as an additional objective, producing a multi-objective genetic program. The primary objective for a genetic programming algorithm is to achieve a favorable score from the deep learning loss function. In prior research, it was determined that certain classes of transformations, such as simple power functions, ratios, differences and counts do not benefit deep neural networks (Heaton, 2016). Therefore, a second objective for a genetic algorithm is to avoid evaluating these transformations. This previous research on algorithms was expanded in the dissertation to determine undesirable expressions so that the AFE algorithm could focus on the most beneficial structures for deep learning.

Genetic programming requires a palette of operators from which to construct expressions. The selection of these operators is critical to a good solution. For example, some problems may benefit by adding the trigonometry functions. However, the trigonometry functions might be unnecessary for other problems. Sometimes problem specific functions are added to the operator set for genetic programming. For the research performed, the operator palette was small:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Protected Division (typically represented as the % symbol)
- Exponent (^)

Neural networks are calculated by applying a weighted sum to each neuron, as demonstrated by Equation 1. Examining the equation reveals that neural networks inherently can multiply and sum. As a result, neural networks do not tend to benefit as much from engineered features involving simple multiplication and addition (Heaton, 2016). Therefore, the algorithm does not expend considerable time on the optimization of constants. It might not be worth the time to discover an engineered feature as complex as the following equation:

$$\boldsymbol{f_e = \frac{3f_1f_2}{2f_3}} \qquad (15)$$

In the above equation, an engineered feature (f_e) was calculated using three of the original features. Because neural networks can perform their own multiplication, it might be possible to simplify to the following equation:

$$\boldsymbol{f_e} = \frac{\boldsymbol{f_1 f_2}}{\boldsymbol{f_3}} \tag{16}$$

The previous two equations are not mathematically equivalent; however, the second equation might be sufficient because the neural network can multiply 3/2 by the engineered feature. Since the research conducted was targeted at deep learning, engineering parts of the feature that the neural network could easily learn during training was avoided. Additionally, other ways to handle constants were determined. Constant coefficients to the engineered features were optimized using gradient descent.

Creating an Efficient Objective Function

Evolutionary algorithms require an objective function to score genomes. These scores determine which genomes will have the opportunity to mate and mutate, thus contributing to the next generation. Because the goal of this dissertation was to engineer features that improve the accuracy of a neural network the objective function is based on an actual neural network that is trained with candidate engineered features (genomes) produced by the genetic programming algorithm. These candidate engineered features are generated from the data set, just like the final set of engineered features recommended by the dissertation algorithm. Because neural networks are computationally expensive to train, this objective function was kept as efficient as possible. To achieve this goal, the neural network had an input neuron for every genome in the population. This makes sense because engineered features ultimately become the inputs to a neural network. Mapping each candidate genome in the population to a neural network input allowed all genomes to be evaluated simultaneously by measuring their feature importance to a trained neural network. The exact process of this objective function will be discussed later in this report.

Experimental Design

To overcome the issues and barriers previously mentioned, a series of experiments were conducted. The results from the first five experiments identified the design characteristics of the genetic programming elements specified in the previous section. Specifically, these experiments showed how to design constraints, measure the success of engineered features, and analyze the population. The experiments conducted are listed here:

- Experiment 1: Limiting the Search Space
- Experiment 2: Establishing Baseline
- Experiment 3: Genetic Ensembles
- Experiment 4: Population Analysis
- Experiment 5: Objective Function Design
- Experiment 6: Automated Feature Engineering

Each of these experiments were measured as described in this chapter.

Measures

Neural networks are a stochastically trained model, and measuring the relative performance of the individual training runs can be difficult. This randomness first occurs because neural networks are initialized with random weights. The randomness continues with the stochastic gradient descent-training algorithm that chooses randomly sampled mini-batches to train the neural network. This randomness in the training process prevents two neural network training runs from producing the same training result. To mitigate the effects of this randomness, multiple neural network cycles were conducted and the average network error was considered.

Feature importance and network error were two primary metrics considered for this dissertation. Feature importance determines how important an individual member of the feature vector is to the predictions of the neural network. Neural network error, the second metric, determines the accuracy of a neural network. Both metrics are described in this section.

Feature importance can be measured for a model through model agnostic calculations and model dependent approaches. A model agnostic calculation measures feature importance purely by querying the model. Internal analysis of the model is not needed. A model agnostic metric works with any type of model, including a neural network, random forest, or support vector machine (SVM). Perturbation feature importance (Breiman, 2001) is a model agnostic calculation technique that was utilized in this research. Additionally, the model dependent approach of analyzing the weights of the neural network was also considered. The perturbation algorithm was chosen for its accuracy and compatibility with deep learning (Olden et al., 2004).

It is also necessary to measure the accuracy of the neural network. The Root Mean Square Error (RMSE) was employed for both classification and regression problems. Although RMSE is sometimes called Root Mean Square Deviation (RMSD), the two terms are equivalent. RMSE/RMSD (McKinney, 2012) is given in the following equation:

$$\mathbf{RMSE} = \mathbf{RMSD} = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}} \qquad (17)$$

The vector *ŷ* represents the actual output vector from the neural network, and the vector *y* represents the expected output from the neural network. The variable *n* represents the number of elements compared. The RMSE reports the error in the same units as the data set. For regression experiments, the RMSE is essentially the average magnitude difference between the predictions and expected outputs. For classification problems, the RMSE is the difference between the predicted and actual probabilities of each class. For a classification problem, the expected probability is 1.0 for the correct class and 0.0 for all other classes.

RMSE values from the same data sets can simply be compared because they are in the same units. However, RMSE values from two different data sets cannot be compared directly. Because the focus of this dissertation is improvement in accuracy for a neural network with a single data set, there is no need to compare across data sets. For this reason, RMSE was chosen as the error evaluation formula for both neural networks and genetic programs. Engineered features were considered successful if they decreased the RMSE (increase accuracy) of a data set by an amount that was deemed statistically significant with a Student’s T-Test.

Experiment 1: Limiting the Search Space

The first experiment continued feature engineering research performed prior to this dissertation (Heaton, 2016). This earlier research demonstrated that neural networks, decision trees, random forests, and gradient boosting machines benefited from different types of engineered feature. Additionally, the research showed that there were several types of engineered feature that were not particularly easy for a neural network to learn on its own. If a neural network could learn to synthesize an engineered feature on its own, adding this feature would not be particularly helpful to the neural network.

The search space of a genetic programming-based algorithm can be narrowed down. The process entails constraining the genomes to expressions similar to expressions that are known to be difficult for the neural networks to synthesize. Heaton (2016) showed the effectiveness of a deep neural network learning to synthesize several types of manually engineered features. The neural network learned single-feature transformations, such as log, polynomial, rational polynomial, power, quadratic, and square root without problem. However, multi-featured transformations were not as simple. The neural network could calculate counts and differences. Rational difference and simple ratios were more difficult for the neural network.

Experimental Design

The first experiment examined the effectiveness of other expression types with neural networks using the same technique as a previous investigation (Heaton, 2016). This method for testing an expression involved generating a data set that used an outcome that was the result of the expression being tested. Randomly sampled values were used as the input (*x*) for these expressions. If the neural network converged to a low RMSE score, then the neural network learned the expression. Furthermore, engineered features in the same format as that expression were more likely to produce more accurate neural networks. RMSE scores were not range normalized because the desire was to see how closely a neural network or genetic program could approximate a function. Because neural networks are stochastic, it is important to perform many experimentation cycles and evaluate the mean and standard deviation of RMSE. For more information on the number of experiment cycles refer to Appendix B, “Hyperparameters.”

Results Format

The results from this experiment were reported in tabular format with the following columns: experiment number, name, expression, standard deviation (SD), minimum error, and mean error. The results from the neural network and genetic programming were reported in different tables. The result format from this experiment for neural networks appeared as Table 2:

Table 2. Experiment 1 results format, neural network and genetic programming

#	Name	Expression	SD	Min Error	Mean Error
1-1	Ratio	$\frac{x_1}{x_2}$	###	###	###
1-2	Ratio Difference	$\frac{x_1 - x_2}{x_3 - x_4}$	###	###	###
…	…	…		…	…

Genetic programs can be taught to approximate expressions, just like neural networks. While genetic programming was not evaluated in the original research, seeing the error for the genetic program gave an indication of the ability of genetic programming to synthesize the expressions. It was expected that genetic programming could easily synthesize any of the expressions and achieve a low error. However, this result was not always the case when the experiments were run. Genetic programming hyperparameters were tuned to achieve this low error. Cases where the genetic programming algorithm did not achieve a low error rate were due to convergence to local optima.

Experiment 2: Establishing Baseline

For the dissertation algorithm to be effective, engineered features from this algorithm needed to enhance neural network accuracy. To measure this performance, several public and synthetic data sets were used to evaluate it. It was necessary to collect a baseline RMSD error of a deep neural network with the data set that received no help from the dissertation algorithm. The neural network topology, or hyperparameters, were determined by the data set dimensions, as described in Appendix B, "Hyperparameters." It was important that the topology included enough hidden neurons so that the neural network could learn the data set with reasonable accuracy.

Experimental Design

The baseline experiment provided a neural network result to compare the results from the dissertation algorithm. Not all data sets were expected to benefit from feature engineering, so it was important to select several real world and synthetic data sets. The list of data sets is provided later in this chapter and described in detail in Appendix D, "Data Set Descriptions" and Appendix E, "Synthetic Data Set Generation." Each neural network was trained until the validation error no longer improved, using an early stopping algorithm.

Results Format

The results from this experiment were reported in tabular format with the following columns: experiment number, name, neural network minimum error, neural network mean error, genetic programming algorithm minimum error, genetic programming algorithm minimum error, genetic programming algorithm mean. Additionally, the t-statistic and p-value from a Student's T-Test is reported between the neural network and genetic programming algorithm. To truly evaluate the effectiveness of the AFE algorithm, it was important to ensure that genetic programming alone could not produce a statistically significant improvement that was equal to or was greater than the dissertation algorithm's gains. The result format from this experiment appeared like Table 3:

Table 3. Experiment 2 result format, neural network baseline

#	Name	Neural Min	Neural Mean	GP Min	GP Mean	T-Statistic	P-Value
2-1	Abalone	###	###	###	###	###	###
2-2	auto-mpg	###	###	###	###	###	###
2-3	Bupa	###	###	###	###	###	###

Experiment 3: Genetic Ensembles

Neural networks and genetic programs can both function as regression models. It is possible to combine models that are trained to accomplish the same objective into ensembles (Dietterich, 2000). A simple blending ensemble takes the output from several models and uses another model type such as a neural network, generalized linear model (GLM), or simple average to combine the results of the member models. It is important to ensure that the AFE algorithm produces results that are better than a simple ensemble of genetic programs. To accomplish this objective, Experiment 3 established a second baseline.

This experiment sought only to use existing genetic programming technology to achieve this baseline. The Encog machine learning framework only supports genetic programming for regression problems. Because of this limitation, only regression data sets were considered for the ensembles experiment. This limitation exists only for

Experiment 3—the AFE algorithm created for this dissertation supported both classification and regression.

Experimental Design

Experiment 3 paralleled Experiment 2 to some degree. However, instead of evaluating individual neural networks, ensembles of neural networks and genetic programs were evaluated for each of the data sets. The ensemble for this experiment was created by training 10 genetic programs to fit the data set with the same inputs and expected output as Experiment 1. A new data set was created that uses these 10 outputs as the input to a neural network. The expected output for the neural network remains the same as the original data set. The structure of this ensemble is shown in Figure 12:

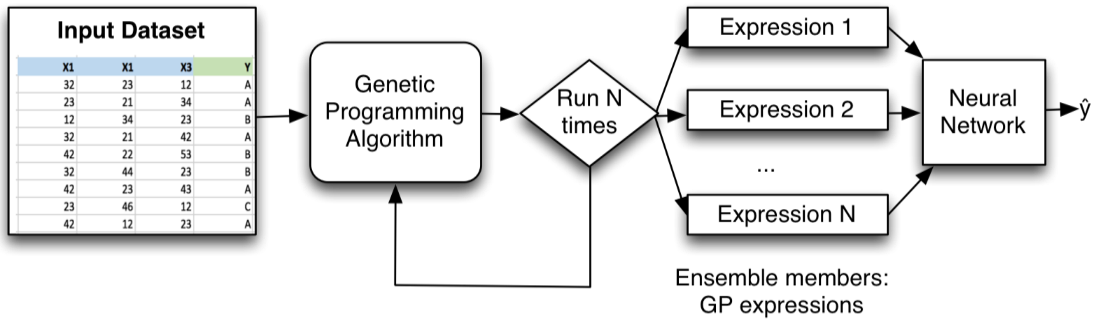


Figure 12. Ensemble of genetic programs for a neural network

Results Format

As with previous experiments, the mean and minimum of the error over several runs was given. The results from this experiment were reported in tabular format with the following columns: experiment name and number, ensemble minimum error, ensemble mean error, and neural network mean error. Like Experiment 2, the results of a Student's T-Test were given between the ensemble and neural network results from Experiment 2, thereby demonstrating the statistical significance of the results. This experiment ensures that a genetic ensemble alone could not achieve the same results as the AFE algorithm that was evaluated in Experiment 6. All error values were reported as RMSE. The result format from this experiment appears like Table 4:

Table 4. Experiment 3-result format, neural network genetic program ensemble

#	Name	Ensemble Min	Ensemble Mean	Neural Mean	T-Statistic	P-Value
3-1	Abalone	###	###	###	###	###
3-2	auto-mpg	###	###	###	###	###
3-3	Bupa	###	###	###	###	###

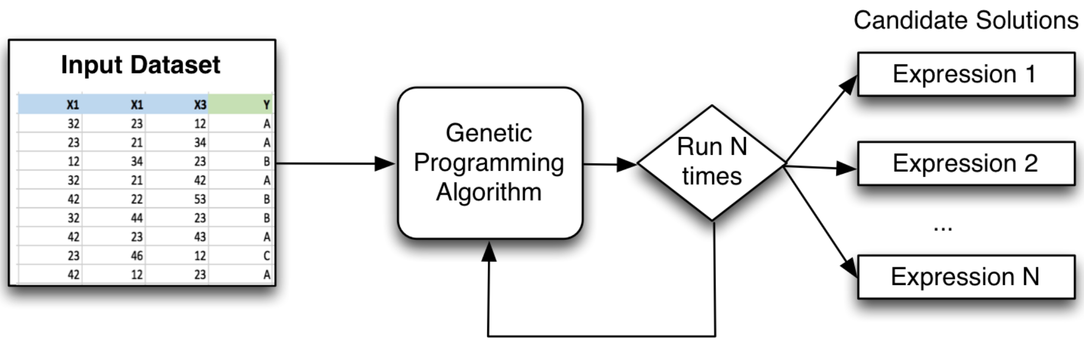
This experiment gave early indications that genetic programming could automatically synthesize information to the data set that the neural network alone could not determine. The results of Experiment 3 can be compared to Experiment 2. For the data sets where the ensemble performed better than the neural network alone, there was the expectation that a genetic programming-based algorithm could engineer a viable feature for further accuracy improvement. If Experiment 3 received a better result on the ensemble than Experiment 2 did with a neural network alone, this result showed that a viable feature has essentially already been engineered by the ensemble. These comparisons were performed as part of Experiment 6.

Experiment 4: Population Analysis

Previous experiments fit genetic programs to function as complete models. These complete models had the potential to use the entire feature vector and be complex. Features that are engineered by human analysis are usually simple combinations of a handful of the original features. The purpose of this experiment was to determine if the complex genetic programming models could be distilled to a representation that provided insight on how to combine elements of the original feature vector into viable, engineered features. These simpler features might perform enough of the calculation to help augment the data for a neural network.

Experimental Design

The input data set allowed the generation of several candidate solutions. This process to generate the ensemble members was the same as in Experiment 3. Figure 13 shows the overall flow for generating the candidate solutions:



X1	X1	X3	Y
32	23	12	A
23	21	34	A
12	34	23	B
32	21	42	A
42	22	53	B
32	44	23	B
42	23	43	A
23	46	12	C
42	12	23	A

Figure 13. Generate candidate solutions

Unlike Experiment 3, the value of *N* was larger. For this experiment, a value of 100 was used. More candidate solutions create more data from which the program can find a pattern. The program searches for the pattern of pairs of input features that are often connected by a common operator, higher in the tree. Figure 14 shows a hypothetical example of a pattern.

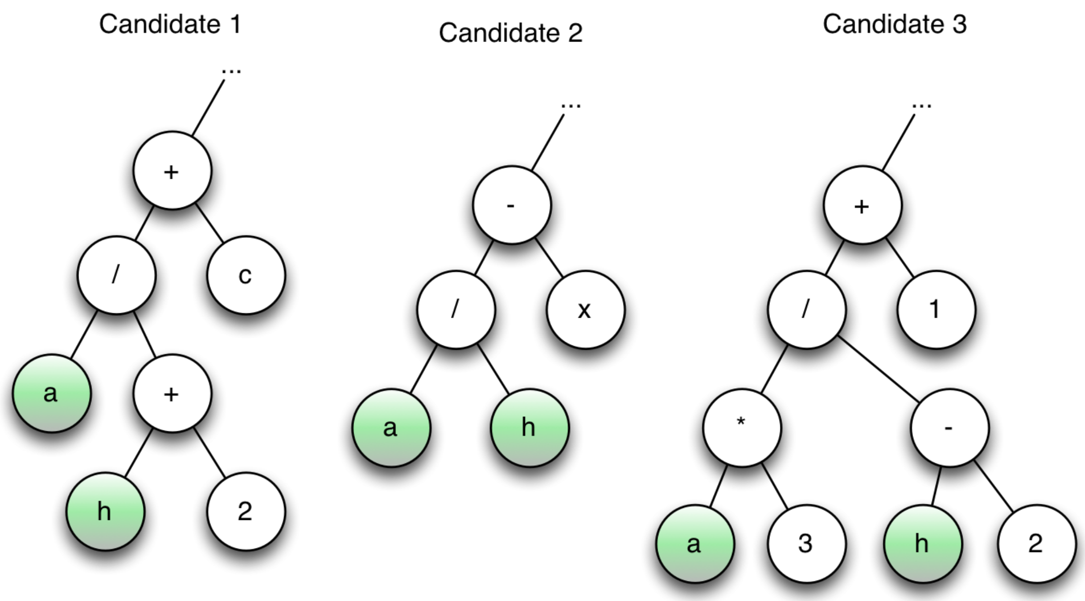


Figure 14. Branches with common structures

This figure shows three branches that might be present on trees from a much larger forest. The three branches always have the features *a* and *h* united by a division operator higher in the branch. Of course, these branches are part of complete trees in the forest of candidate solutions. Partial branches are shown for clarity and brevity. The fact that *a* and *h* often occur together in a ratio might indicate that final engineered features use these two in a ratio. This information could allow the construction of the features outright. It could also narrow the search space for a given data set.

Results Format

The results from this experiment were reported in tabular format with the following columns: data set number, name, and the patterns found. Patterns were reported in a format such as “a/b”, “a*b”, or “(a+b)/c”. The result format from this experiment is like Table 5:

Table 5. Experiment 4 results format, patterns in genetic programs

#	Name	Patterns
4-1	Auto MPG	(x1/x2) (x2*x3) ((x1*x2)+x1)
4-2	Wisconsin Breast Cancer	(x2*x3)
...	…	...

Experiment 5: Objective Function Design

Evolutionary algorithms work through a process that is based on natural selection (Holland, 1975). To select from among the genomes, it is necessary to evaluate the effectiveness of the candidate-engineered features relative to each other. Several algorithms determine the feature importance of the neural network. For this research, a feature-ranking algorithm should be relatively stable, despite the stochastic nature of neural networks. Feature ranking stability was measured by the degree to which multiple neural networks produced the same ranking of features for the same data set.

The goal of this experiment was to determine an effective feature ranking algorithm for the dissertation research. The two main considerations for feature ranking were the following:

- Which feature-ranking algorithm is the most stable?
- How far should a neural network be trained?

It was desirable to keep the objective function as computationally inexpensive as possible. The faster the objective function executed, the larger a search space to be covered. To evaluate feature ranking, it was not necessary to train the neural networks to their maximum accuracy. Instead, it was crucial to train the neural network to the point that the ranking algorithm can give a stable assessment of the relative importance of each of the features. This experiment provided answers to both concerns.

Experimental Design

This experiment evaluated the ranking of the original feature vector for each of the data sets considered for this dissertation. Each data set was trained multiple times with its feature ranking evaluated at each training iteration. A validation holdout allowed the training algorithm to know the stopping point. Once the validation set's error no longer improved, training stopped. The percent validation improvement was reported for the iteration where the feature ranking first stabilized to the same order as the final feature ranking. The idea was that training could be stopped at the stage where the validation error improvement hit that point, and a reasonably accurate feature rank could be determined.

Results Format

Each of the real world (non-synthetic) data sets were evaluated, and the minimum validation set improvements were reported. Additionally, the final feature rank was also reported. The results appear like Table 6:

Table 6. Experiment 5 results format, evaluating feature ranking

#	Name	Rank	Weight	Permutation
5-1	Auto MPG	x3,x2,x1,x10,x11	###	###
5-2	Wisconsin Breast Cancer	x3,x2,x4,x10,x1	###	###
...	...	...	...	...

Experiment 6: Automated Feature Engineering

The final experiment conducted brought the results of the previous five experiments into the AFE algorithm, thereby accomplishing the objective of this dissertation. The goal of Experiment 6 was to compare each data set. It showed the neural network validation error next to the same neural network that had been augmented with features that the dissertation algorithm engineered.

Experimental Design

This experiment was conducted similarly to Experiment 1, except that the neural networks were augmented with features that were engineered from each of the data sets. The dissertation algorithm's effectiveness was evaluated by comparing the neural network's accuracy on the augmented data set with the baseline result in Experiment 6. Comparing the results of the dissertation algorithm to the Experiment 1 baseline showed that some of the data sets had improved results when augmented with engineered features.

The feature-engineering algorithm's development was guided by Experiments 1, 3, 4, and 5. Experiment 2 established a baseline for comparing the algorithm. Experiment 6 underwent several iterations as the algorithm was enhanced and its performance measured. The final algorithm, measured by Experiment 6, is given in Chapter 4, "Results." The framework for fitting the experiment results into the preliminary algorithm is given by Figure 15.

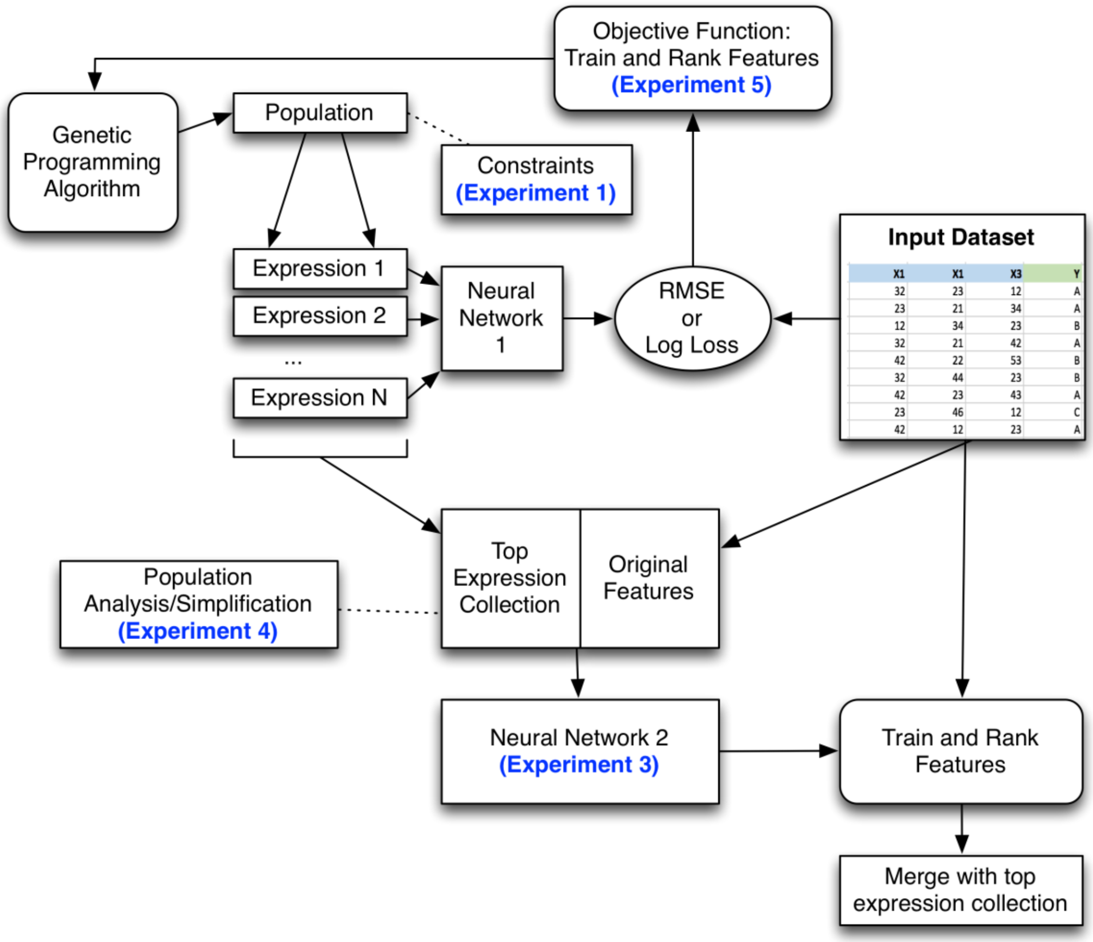
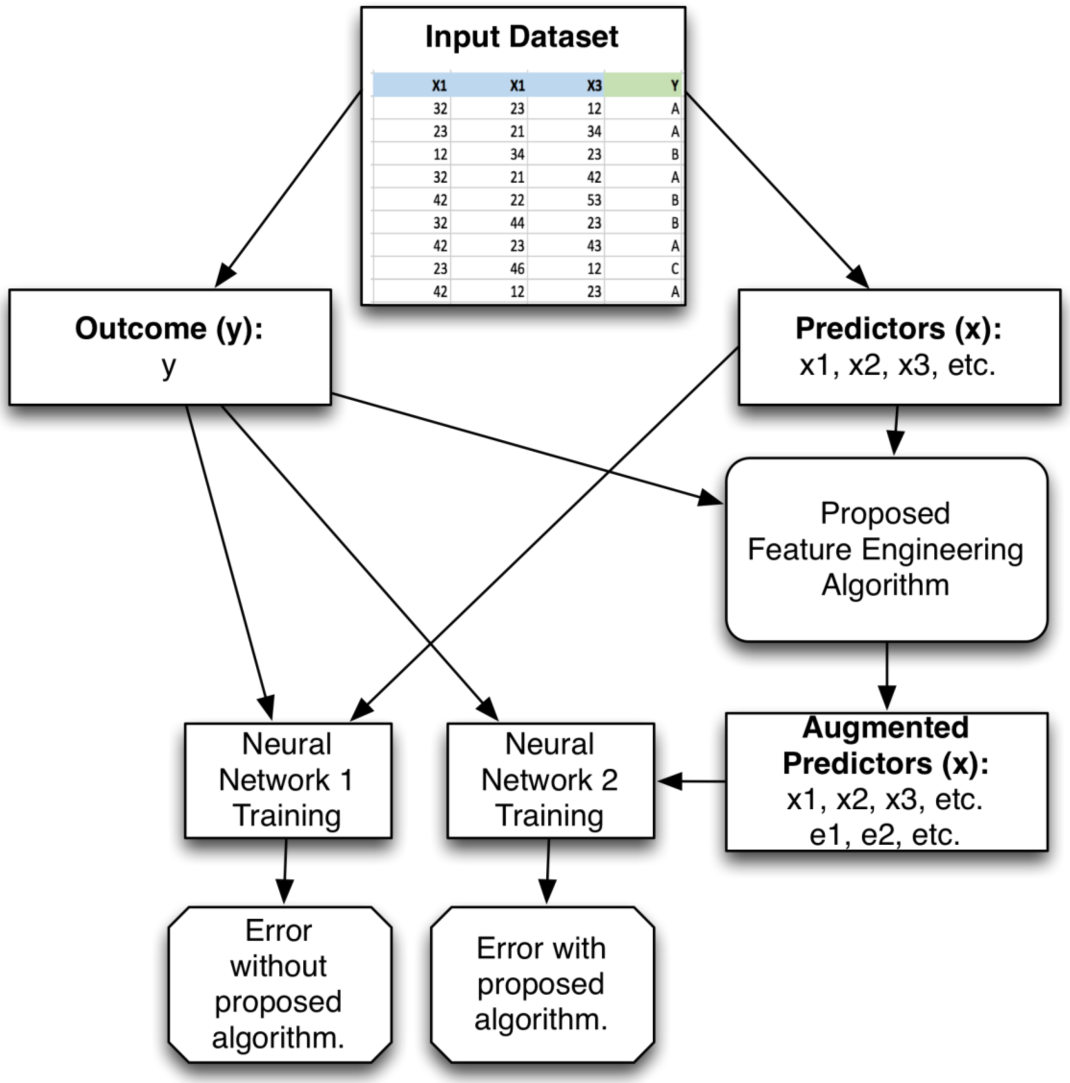


Figure 15. High-level overview of AFE algorithm

This algorithm used two neural networks. Neural Network 1 evaluated a population of potential engineered features. This population was constrained using information learned from Experiment 1. The objective function came directly from Experiment 5. Experiment 4 provided information to refine the candidate expressions from the population into potential engineered features. These engineered features were ranked and evaluated using Neural Network 2 and examined to join a list of top engineered expressions. The above abstract flow chart evolved into the final AFE algorithm given in the next chapter. The goal was to engineer features that could boost the baseline accuracy (Experiment 1) of the data sets.

Two neural networks were trained. The first neural network was trained on the original predictors (*x*) and the second on the augmented feature vector. Both were trained to produce the outcome (*y*) that this algorithm did not alter. Figure 16 shows the setup for the final evaluation.


Input Dataset
Outcome (y):
y
Predictors (x):
x1, x2, x3, etc.
Proposed Feature Engineering Algorithm
Augmented Predictors (x):
x1, x2, x3, etc.
e1, e2, etc.
Neural Network 1 Training
Neural Network 2 Training
Error without proposed algorithm.
Error with proposed algorithm.

X1	X1	X3	Y
32	23	12	A
23	21	34	A
12	34	23	B
32	21	42	A
42	22	53	B
32	44	23	B
42	23	43	A
23	46	12	C
42	12	23	A

Figure 16. Dissertation algorithm evaluation

This figure closely resembles Figure 11. Essentially, the program forked the input data set to be fed to two neural networks for training. The leftmost neural network trained without the AFE algorithm, and the neural network on the right trained with the algorithm. The objective was to determine if the neural network with augmented predictors could perform better than a neural network with only the original features.

Results Format

The results of this experiment showed which features could be engineered for each of the data sets in the dissertation. The results contained the following columns: experiment number, data set name, neural network mean RMSE, auto/augmented neural network RMSE, and improvement by using the algorithm. The neural network error column reports the same values as Experiment 1 because this experiment compares the dissertation algorithm to the neural network baseline. Table 7 shows the anticipated format for the results of Experiment 6:

Table 7. Experiment 6 results format, engineered feature effectiveness

#	Name	Neural Mean	Auto Mean	Difference	Change Percent
6-1	Abalone	###	###	###	###
6-2	auto-mpg	###	###	###	###
6-3	Bupa	###	###	###	###

It was assumed that the AFE algorithm introduced in this dissertation would not improve accuracy for each data set. The statistical significance of the change in this table was evaluated for all data sets that show a negative percent change. Additionally, this change was compared against the change in accuracy observed for the genetic programming algorithms in Experiment 1 and ensembles in Experiment 2. This result demonstrated that the AFE could provide more accuracy than genetic programming alone.

Real World Data sets

The success of an AFE algorithm was ultimately measured by its success in the generation of actual features for a real-world data set. The following were the primary sources for real-world data sets:

- UCI Machine Learning Repository (Newman & Merz, 1998)
- PROBEN1 Datasets (Prechelt, 1994)

All real-world data sets came from the UCI machine learning repository. Data sets that had the following attributes were favored:

- No image or audio data sets
- At least 10 numeric (continuous) features
- Features should be named, such as measurements, money or counts

The following UCI data sets were used for this research:

- Abalone Data Set (Abalone)
- Auto MPG Data Set (Auto-mpg)
- Liver Disorders Data Set (Bupa)
- Covertype Data Set (Covtype)
- Credit Approval Data Set (Crx)
- Forest Fires Data Set (Forestfires)
- Glass Identification Data Set (Glass)
- Heart Disease Data Set (Heart)

- Hepatitis Data Set (Hepatitis)
- Horse Colic Data Set (Horse)
- Housing Data Set (Housing)
- Pima Indians Diabetes Data Set (Pima)
- Soybean (Large) Data Set (Soybean)
- Breast Cancer Wisconsin P (Diagnostic) (WcBreastP)
- Breast Cancer Wisconsin D (Diagnostic) (WcBreastD)
- Wine Data Set (Wine)

Prechelt (1994) introduced the PROBEN1 collection of data sets. This collection contained 13 standardized data sets from the UCI repository. The paper that presented PROBEN1 had neural network benchmark results. Additionally, the PROBEN1 paper has over 900 citations, many of which publish additional neural network results on these data sets. These characteristics made the PROBEN1 data sets good candidates for comparison in this study. For more information on how these UCI data sets were preprocessed for this dissertation refer to Appendix D, “Data Set Descriptions.”

Synthetic Data sets

Not all data sets saw increased accuracy from engineered features, especially if the underlying data did not contain relationships that feature engineering can expose. Thus, it was necessary to create data sets that were designed to include features that are known to benefit deep neural networks. A program was created that generated data sets that contained outcomes that were designed to benefit from feature engineering of varying degrees of complexity. It was necessary to choose engineered features that the deep neural networks could not easily learn for themselves. The goal was to engineer features that helped the deep neural network—not features that would have been trivial for the network to learn on its own.

In previous research, Heaton (2016) formulated a simple way to learn the types of features that benefit a deep neural network. Training sets were generated in which the expected output was the output of the engineered feature. If the model could learn to synthesize the output of the engineered feature, then adding this feature would not benefit the neural network. This process was like the common neural network example of teaching itself to become an XOR operator. Because neural networks could easily learn to perform as XOR operators, the XOR operation between any two original features would not make a relevant engineered feature. For more information on how the synthetic data sets used in this dissertation were created, refer to Appendix E, "Synthetic Data Set Generation."

Resources

The hardware and software components necessary for this dissertation were all standard, readily available, common, and off-the-shelf personal computer system components and software. Two quadcore Intel I7 Broadwell-equipped machines with 16 gigabytes of RAM were available for this research. These systems performed many computations needed to support this research. In addition, the author made considerable use of Amazon EC2/AWS. An AWS c3.8xlarge instance type with 32 processing cores and 60 gigabytes of RAM was the primary resource. Funds were also budgeted to purchase AWS cloud time. Given the computational complexity of the AFE algorithm, a high-end compute optimized AWS instance was invaluable. There were no financial costs to Nova Southeastern University for this research.

The Java programming language (Arnold, Gosling, & Holmes, 1996) served as the programming language to complete this research. The Java 8 version (JDK 1.8) provided the specific implementation of the programming language. Furthermore, Python 3.5 (Van Rossum, 1995) was used in conjunction with Scipy (Jones, Oliphant, Peterson, & al., 2001), scikit-learn (Pedregosa et al., 2011), TensorFlow (Abadi et al., 2016) for additional deep learning experimentations. The Python machine learning packages were used to compare select neural networks and feature combinations with the Encog library.

Encog version 3.4 (Heaton, 2015) provided the deep learning and genetic programming portions of this research. Encog gives extensive support for both deep learning and genetic programming. Additionally, Encog is available for both the Java and C# platforms. The author of this dissertation wrote much of the code behind Encog and has extensive experience with the Encog framework. For a complete list of third party libraries used in this dissertation, refer to Appendix C, “Third Party Libraries.”

Summary

This dissertation leveraged genetic programming to create an algorithm that could engineer features that increased the accuracy of a deep neural network. Not all data sets contained features that could be engineered into a better feature vector for the neural network. Thus, it was important to use several different data sets to evaluate the AFE algorithm. The effectiveness of the algorithm was determined by evaluating the change in error between two neural networks—one had access to the algorithm’s engineered features and the other did not.

The algorithm was created by combining the knowledge gained from five experiments. The sixth experiment performed a side-by-side benchmark between data sets augmented with features engineered by the algorithm and those that were not. The effectiveness of the algorithm was measured by the degree to which the error decreased for the feature-engineered data set, when compared to an ordinary data set.

The five experiments evaluated how to leverage different aspects of genetic programming for neural network feature engineering. Expressions that were beneficial to neural networks were explored. Objective function design was examined. Neural network feature ranking was optimized for the quickest results. Ensembles could detect data sets that benefitted the most from feature engineering. Information gained from these experiments guided the algorithm design.

For reasons of confidentiality, the source code will not be publicly distributed prior to formal publication of the dissertation report. At that point, the source code necessary to reproduce this research will be placed on the author's GitHub[1] repository. To see the complete implementation in Java source code, refer to Appendix H, "Dissertation Source Code Availability." Upon completion of the project, the final dissertation report will be distilled and submitted as an academic paper to a journal or conference.

[1] http://www.github.com/jeffheaton

Chapter 4

Results

Introduction

The five experiments were conducted, and data from their results were compiled. These results helped to guide the creation of the final algorithm that automatically engineers features for deep neural networks. This algorithm was developed and improved by rerunning and optimizing the neural network and genetic programming hyperparameters for the neural and genetic programming algorithms. The final set of hyperparameters that were used across all experiments is provided in Appendix B, "Hyperparameters." The complete source code for all experiments and the final algorithm is available at the author's GitHub account. For more information on obtaining the source code refer to Appendix H, "Dissertation Source Code Availability."

Finally, the sixth experiment was conducted to demonstrate the degree to which the AFE algorithm increased the accuracy of deep neural networks. This chapter begins with a summary of the results obtained from each of the five experiments. The complete detail is provided in Appendix G, "Detailed Experiment Results." Next, the final AFE algorithm is defined and the rational for its design explained. Finally, this chapter concludes with the results of Experiment 6, where the effectiveness of the AFE algorithm is demonstrated.

Experiment 1 Results

The purpose of the first experiment was to continue the research of the author (Heaton, 2016) and empirically demonstrate which types of features have the greatest potential to increase a deep neural network's predictive accuracy. This result was accomplished by finding several expression types that neural networks cannot easily learn independently. Because the neural networks cannot easily learn these expression types on their own, adding them to the input feature vector has the potential of increasing the neural network accuracy. Of course, the expression must expose some meaning for the data set, such as an interaction between two features. Thus, an increase in accuracy was not observed for every data set investigated.

Experiment 1 investigated how well neural networks and genetic programming could fit to a synthetic data set made up of several mathematical expression types. A lower RMSE indicates that the neural network or genetic program could easily replicate the expression without the need to engineer it. A higher RMSE indicates that the neural network or genetic program had a more difficult task replicating the expression. The fact that a neural network has a more difficult time replicating a given expression generally indicates an expression type that might be a valuable type of engineered feature for the neural network. For more information about how the synthetic expression data set was generated, refer to Appendix E, "Synthetic Data Set Generation."

The results from Experiment 1 are presented in two tables. Table 8 shows the results that were observed from fitting a neural network to the synthetic expressions dataset. Because neural networks are stochastic, it is important to perform many experimentation cycles and evaluate the mean and standard deviation of accuracy. Similarly, Table *9* shows the results that were observed from fitting a genetic program to the expressions data set with error measured in RMSE:

Table 8. Experiment 1 neural network results for select expressions

#	Name	Expression	SD	Min Error	Mean Error
1-1	Difference	$x_1 - x_2$	0.3767	0.0006	0.3152
1-2	Log	$\log(x_1)$	0.3314	0.0419	0.4762
1-3	Polynomial	$8x_1^2 + 5x_1 + 1$	1.4432	0.0585	2.5
1-4	Polynomial2	$5x_1^2x_2^2 + 4x_1x_2 + 2$	0.392	0.1534	0.7735
1-5	Power	x_1^2	0.0928	0.0216	0.2414
1-7	Ratio	$\frac{x_1}{x_2}$	0.0181	27.3997	27.4702
1-8	Ratio Difference	$\frac{x_1 - x_2}{x_3 - x_4}$	0.0042	22.973	22.9831
1-6	Ratio Polynomial	$\frac{1}{8x_1^2 + 5x_1 + 1}$	0.0586	9.9919	10.231
1-9	Ratio Polynomial2	$\frac{1}{5x_1^2x_2^2 + 4x_1x_2 + 2}$	0.0722	0.0287	0.1558
1-10	Square Root	$\sqrt{x_1}$	0.0975	0.0075	0.1703

Table 9. Experiment 1 genetic program results for select expressions

#	Name	Expression	SD	Min Error	Mean Error
1-11	Difference	$x_1 - x_2$	0.1712	0	0.0624
1-12	Log	$\log(x_1)$	0.1221	0.0229	0.2
1-13	Polynomial	$8x_1^2 + 5x_1 + 1$	0.4141	0.0002	0.9434
1-14	Polynomial2	$5x_1^2x_2^2 + 4x_1x_2 + 2$	0.252	0.2	0.7008
1-15	Power	x_1^2	0.0348	0	0.0057
1-16	Ratio	$\frac{x_1}{x_2}$	6.5254	0	2.4261
1-17	Ratio Difference	$\frac{x_1 - x_2}{x_3 - x_4}$	0.0158	22.906	22.9732
1-18	Ratio Polynomial	$\frac{1}{8x_1^2 + 5x_1 + 1}$	0.2091	7.7973	7.8369
1-19	Ratio Polynomial2	$\frac{1}{5x_1^2x_2^2 + 4x_1x_2 + 2}$	0.0406	0.0812	0.1587
1-20	Square Root	$\sqrt{x_1}$	0.038	0	0.0373

It is evident from Table 8 that neural networks are not particularly adept at synthesizing the expression types of rational difference and ratio. It can also be observed that genetic programming can often synthesize several features perfectly, such as the difference, power, ratio, and square root. The genetic programming algorithm (Table 9) did sometimes attain a perfect 0 RMSE (as evident by the minimum column), whereas the neural networks (Table 8) never achieved a perfect 0 RMSE. Yet the fact that rows where the minimum did hit zero have a higher mean score indicates that genetic programming does have the potential to engineer features that are not easily synthesized by neural networks.

There were two primary conclusions from Experiment 1. First, the genetic programming algorithms are prone to settling into local optima. Second, genetic programs can synthesize certain expressions that neural networks are not able to synthesize. This result is consistent with prior research. Furthermore, previous investigations are extended (Heaton 2016) in this dissertation with the demonstration of genetic programs and their ability to overcome the limitations of neural networks, for some data sets.

These observations influenced the algorithm design in two ways. First, the AFE algorithm aggregated many genetic programming solutions together to mitigate the effects of local optima. Second, the search space of potential expressions was biased to avoid expression types that neural networks could synthesize without the need for feature engineering.

Experiment 2 Results

The purpose of the second experiment was to create a baseline of neural network and genetic programming performance over several real world and synthetic data sets. This baseline was compared against the AFE algorithm in Experiment 6. Improving the accuracy that neural networks attained in Experiment 2 was ultimately the objective of Experiment 6 as it evaluated the effectiveness of the AFE algorithm. Although the focus of Experiment 2 is to provide baseline neural network results, genetic programming results were also obtained. These results were useful to ensure that genetic programming alone was not sufficient to improve the neural network accuracy. Furthermore, classification problems were not considered with genetic programming because Encog does not currently support genetic programming classification. This Encog limitation does not affect the AFE algorithm's ability to handle classification problems because regression genetic programming is the only part that the dissertation algorithm used internally. The baseline results are presented in Table 10:

Table 10. Experiment 2 baselines for data sets (RMSE)

#	Name	Neural Min	Neural Mean	GP Min	GP Mean	T-Statistic	P-Value
2-1	Abalone	2.0992	2.6523	2.2149	**2.5843**	1.4487	0.149
2-2	auto-mpg	2.2207	3.3376	3.0231	5.4287	-9.3793	0
2-3	Bupa	0.4005	0.4552	0.4032	0.4746	-4.0954	0.0001
2-4	Covtype	0.3382	0.3999				
2-5	Crx	0.3062	0.3593	0.3168	0.3877	-7.9337	0
2-6	Forestfires	0.057	0.0608				
2-7	Glass	0.2743	0.3377				
2-8	Heart	0.3016	0.3378				
2-9	Hepatitis	0.3973	0.4576	0.3922	**0.4422**	4.8029	0
2-10	horse-colic	0.3248	0.366				
2-11	Housing	0.0445	0.0447				
2-12	Iris	0.034	0.3182				
2-13	Pima	0.3791	0.4164	0.4019	8.36E+15	-1.0051	0.3161
2-14	soybean_large	0.068	0.1936				
2-15	wcbreast_wdbc	0.0803	0.0989	0.1827	0.3033	-38.5506	0
2-16	wcbreast_wpbc	0.293	0.3812	0.4049	0.436	-13.1433	0

The first column provides the experiment and test number. The second and third columns give the neural network minimum and mean across all experiment cycles. The fourth and fifth columns state the genetic programming algorithm minimum and maximum RMSE scores across all experiment cycles. Finally, the sixth and seventh columns indicate the results of a Student's T-Test to investigate the statistical significance of differences between the effectiveness of a neural network and genetic program for the given data set. The null hypothesis is that genetic programming and neural networks perform the same for a given data set. A p-value that is less than 0.05

indicates that the null hypothesis can be rejected. Values in column 6 that are bolded indicate a genetic programming result that was superior to the neural network result.

All values are given in RMSE, which means that the errors reported are in the same unit as the data set's target variable. Thus, RMSE errors between two rows cannot be compared. There were only two data sets where genetic programming performed better than the neural network, and both were relatively small improvements. The improvement noted in the Abalone data set was not statistically significant, as indicated by the p-value above 0.05. Genetic programming did produce a small but statistically significant improvement for the hepatitis data set. Experiment 6 demonstrated this result with the evaluation of the auto feature engineering algorithm's effectiveness on this data set. All the other data sets did not perform as well with genetic programming as they did with neural networks. Many of these cases of poorer genetic programming performance were statistically significant, as evident by their p-values. The fact that the AFE algorithm increased the accuracy of some of these data sets is further evidence that the algorithm has capabilities beyond traditional genetic programming.

Experiment 3 Results

Experiment 3 demonstrated that neural networks and genetic programs could function together as ensembles. In some cases, an ensemble of genetic programs and a neural network could provide greater accuracy than a neural network alone. Experiment 3 can be considered an early prototype of the final AFE algorithm. However, Experiment 3 did not create an algorithm. Experiment 3 simply used off-the-shelf genetic programming technology to fit a series of genetic programs to a data set. An off-the-shelf deep neural network is used to ensemble the output from the genetic programs into a final output. The accuracy of that final output is evaluated and reported in Table 11:

Table 11. Experiment 3, Ensemble, GP & neural network results

#	Name	Ensemble Min	Ensemble Mean	Neural Mean	T-Statistic	P-Value
3-1	Abalone	2.0783	2.6275	2.6523	0.3819	0.7029
3-2	auto-mpg	2.1895	3.7463	3.3376	-1.3475	0.1794
3-3	Bupa	0.3957	0.4503	0.4552	0.9474	0.3446
3-4	Crx	0.2966	0.3539	0.3593	1.2481	0.2135
3-5	Hepatitis	0.4185	0.4586	0.4576	-0.323	0.747
3-6	Pima	0.3762	0.4122	0.4164	0.9676	0.3344
3-7	wcbreast_wdbc	0.0803	0.0979	0.0989	0.367	0.714
3-8	wcbreast_wpbc	0.2656	0.3869	0.3812	-1.0239	0.3071

The first column indicates the experiment and test number. The second column provides the data set name. The third and fourth columns give the minimum and mean RMSE scores attained by the ensemble over the experiment cycles. The fifth column states the neural network mean RMSE score attained over the experiment cycles.

An ensemble of genetic programs yielded small accuracy increases, when compared with a singular neural network in several of the cases in Table 11. The AFE algorithm built upon this result to provide additional accuracy.

Experiment 4 Results

Experiment 4 analyzed many genetic programs to find common patterns for individual data sets. As Experiment 2 demonstrated, genetic programming tends to fall into local optima. Thus, genetic programming algorithms will not always return the same result for the same data set. Many runs of a genetic programming algorithm will return many different solution outcomes. Experiment 4 demonstrated that patterns can be found in these solutions that the genetic programming algorithm returned.

To facilitate Experiment 4, a simple pattern-finding algorithm was produced that accepted a genetic program and returned all combinations of input variables and operators that it could find. This algorithm was implemented as a recursive function that operated on a genetic program represented as a tree. The input of this function was a genetic programming tree, and the output was several strings that represented every combination of input variable and operator that was found. Every combination of two or more variables linked by one or more operators were returned. Constant nodes were not considered.

The following listing shows this algorithm, which was invoked by calling the *find* function:

```
def find_at_node(parent):
  vars_found.clear()
  found_expression = render_at_node(parent)

  if len(vars_found.size >= 2:
    if patterns.contains(foundExpression):
      patterns.get(foundExpression).increase();
    else:
      patterns.append(foundExpression,t);

  for tn in parent.children:
    find_at_node(child)


def find(prg):
  root = prg.root
  find_at_node(root)
  return patterns
```

The above pseudocode was implemented as the class *FindPatternsGP* in the Java source code provided with this dissertation. The function *render_at_node* was not shown for brevity, as it simply pruned constants and rendered tree structures to infix expressions. To see the complete implementation in Java source code, refer to Appendix H, “Dissertation Source Code Availability.”

The following listing shows how the above algorithm would encode several expressions:

```
Expression: (x*y)
  (x*y)

Expression: (x*y)*z
  ((x*y)*z)
  (x*y)

Expression: ((y/1)+(x*2))/1
  (x+y)
```

These expressions can be thought of as looking at a tree representation and pruning all constant nodes. The reason for this is to find common operator patterns, across many expressions. The constants are not considered for these patterns. The results above are what remain from such a pruning. Table 12 shows the most common patterns found by Experiment 4 using the above algorithm:

Table 12. Experiment 4, patterns from genetic programs

#	Name	Rank	Pattern
4-1	Abalone	0.02	(diameter-shucked)
		0.02	(-(diameter)*shucked)
		0.02	((height+shell)*sex)
		0.01	(((diameter-shucked)-shucked)+shell)
		0.01	((diameter-shucked)-shucked)
4-2	Auto-mpg	0.03	(acceleration-(cylinders-origin))
		0.01	(cylinders-origin)
		0.01	((-(cylinders)+origin)+acceleration)
		0.01	0.0100:(-(cylinders)+origin)
		0.01	0.0100:((cylinders+horsepower)+acceleration)
4-3	Bupa	0.04	((((-(alkphos)*drinks)*(alkphos+gammagt))-mcv)+gammagt)
		0.02	(mcv/((((((-(alkphos)*drinks)*(alkphos+gammagt))-mcv)+gammagt)*gammagt))
		0.01	(sgpt^sgot) (((((-(alkphos)*drinks)*(alkphos+gammagt))-mcv)+gammagt)*gammagt)
		0.01	(((-(alkphos)*drinks)*(alkphos+gammagt))-mcv)
4-4	Crx	0.02	(a12+a9)
		0.01	(a11+a9)
		0.01	((-(a4)+a2)^-(a9))
		0.01	(-(a4)+a2)
		0.01	(a9-((a10/a10)/a9))
4-5	Hepatitis	0.04	(spiders-sex)
		0.01	-((sex/((malaise^(spiders-sex))-albumin)))
		0.01	(sex/((malaise^(spiders-sex))-albumin))
		0.01	((malaise^(spiders-sex))-albumin)
		0.01	(malaise^(spiders-sex))
4-6	Pima	0.03	(insulin^diastolic)
		0.02	(age+bmi)
		0.02	((age-bmi)-diastolic)
		0.02	(age-bmi)
		0.01	(age^(diabetes/-(diabetes)))
4-7	WcBreastP	0.03	(mean_concave_points+worst_perimeter)
		0.01	(-(se_area)^-(((mean_concave_points +worst_perimeter)*mean_concavity)))
		0.01	-(((mean_concave_points

			+worst_perimeter)*mean_concavity))
		0.01	((mean_concave_points +worst_perimeter)*mean_concavity)
		0.01	((se_concavity^((-(worst_radius) *worst_concave_points)*mean_concavity)) ^(-((se_area^se_symmetry))/ -(se_symmetry)))
4-8	WcBreastD	0.02	((se_concavity^worst_concavity) -(worst_texture/((se_compactness+se_texture) -mean_radius)))
		0.02	(se_concavity^worst_concavity)
		0.01	(worst_texture/((se_compactness+se_texture) -mean_radius))
		0.01	((se_compactness+se_texture)-mean_radius)
		0.01	(se_compactness+se_texture)

This table was generated by running a genetic programming algorithm 100 times on each data set. This process produced 100 different candidate solution expressions. These algorithms were run against the pattern-finding algorithm. As the above table demonstrates, several common patterns emerged for each data set.

The fact that common patterns exist in the candidate genetic programming was very important to the design of the AFE algorithm that was created for this dissertation. The AFE algorithm used many genetic programs to derive the final set of engineered features that it recommended. Results from Experiment 4 and the algorithm presented in this section were important to the design of the final AFE algorithm.

Experiment 5 Results

Experiment 5 provided results to determine the most effective means of ranking the relative importance of features in a neural network's feature vector. The goal of Experiment 5 was to evaluate several feature ranking algorithms to determine if the rankings stabilized before the neural network was fully trained. Because the automated feature ranking algorithm must evaluate many features, considerable time could be saved by not fully training the neural networks that are being used to judge candidate engineered features.

Two feature ranking algorithms were considered for Experiment 5. The first was the perturbation ranking algorithm (Breiman, 2001). The second was a weight analysis algorithm, based on research performed by Garson (1991), that evaluated the weights between the input and first hidden layers. A neural network was trained, using an early stopping strategy, to measure the stability of feature ranking during training.

Unfortunately, neither feature ranking algorithm was stable at any phase of training, on any data set, or even a few steps before training was stopped. This result can be observed by examining the ranking for several training steps for the auto MPG data set, as demonstrated below:

```
Epoch #1, Train Error:13.927148,
  Perm(0):2,8,7,0,1,5,3,4,6,
  Weight(0):4,1,7,5,0,2,6,3,8,
  Validation Error: 23.959034, Stagnant: 0
Epoch #2, Train Error:26.961522,
  Perm(0):3,8,6,4,7,1,2,0,5,
  Weight(0):4,7,1,5,0,2,6,8,3,
  Validation Error: 23.959034, Stagnant: 0
Epoch #3, Train Error:25.796014,
  Perm(0):3,1,8,7,0,6,4,5,2,
  Weight(0):4,7,1,5,0,6,2,8,3,
  Validation Error: 23.959034, Stagnant: 0
Epoch #4 Train Error:19.821996,
  Perm(0):1,3,8,7,0,4,6,2,5,
  Weight(0):4,7,1,0,5,6,8,2,3,
Validation Error: 23.959034, Stagnant: 0
  Epoch #5 Train Error:16.163487,
  Perm(0):3,8,1,7,4,0,6,2,5,
  Weight(0):4,7,0,1,5,6,8,2,3,
  Validation Error: 23.959034, Stagnant: 0
```

```
Epoch #6 Train Error:24.637161,
  Perm(0):3,8,7,1,4,0,6,5,2,
  Weight(0):4,7,0,1,6,5,8,2,3,
  Validation Error: 23.959034, Stagnant: 0
Epoch #7 Train Error:26.744201,
  Perm(0):1,3,8,7,6,0,4,2,5,
  Weight(0):4,0,7,6,1,8,5,2,3,
  Validation Error: 23.357838, Stagnant: 0
Epoch #8 Train Error:26.495994,
  Perm(0):3,8,1,7,6,0,4,2,5,
  Weight(0):4,0,7,6,8,1,5,2,3,
  Validation Error: 23.357838, Stagnant: 0
Epoch #9 Train Error:35.010211,
  Perm(0):1,3,8,0,7,4,6,2,5,
  Weight(0):4,0,7,6,8,5,1,2,3,
  Validation Error: 23.357838, Stagnant: 0
Epoch #10 Train Error:24.311009,
  Perm(0):3,1,0,8,7,4,6,2,5,
  Weight(0):4,0,6,8,7,5,1,2,3,
  Validation Error: 23.357838, Stagnant: 0
```

The feature ranking shuffled as the individual training steps were executed. One can conclude from Experiment 5 that it was difficult to evaluate partially trained networks to obtain a definitive ranking early in training. Early stabilizing feature ranking algorithms remains an open area of research. While feature ranking was an important aspect of the AFE algorithm, several steps were taken to mitigate the effects of unstable feature ranking algorithms.

AFE Algorithm Design

Experiments 1 through 5 provided the information that was needed to design an AFE algorithm. Once these experiments were complete, it was possible to design this algorithm. This section provides the design of the AFE algorithm that Experiment 6 will evaluate.

Experiments 1 through 5 yielded the following observations:

- Larger data sets take considerable time for the GP to fit.
- Feature ranking is unstable until neural network training has finalized.
- GP is very prone to finding many local optima.
- GP produces many different candidate solutions with pronounced sub-patterns.
- Coefficients will vary greatly in solutions created by genetic algorithms.

The following sections describe the AFE algorithm and how these observations influenced its design.

High Level Overview

This section presents a high-level overview of the AFE algorithm that was developed for this dissertation and is presented in Figure 17. Subsequent sections will further elaborate on the design of the components summarized in Figure 17.

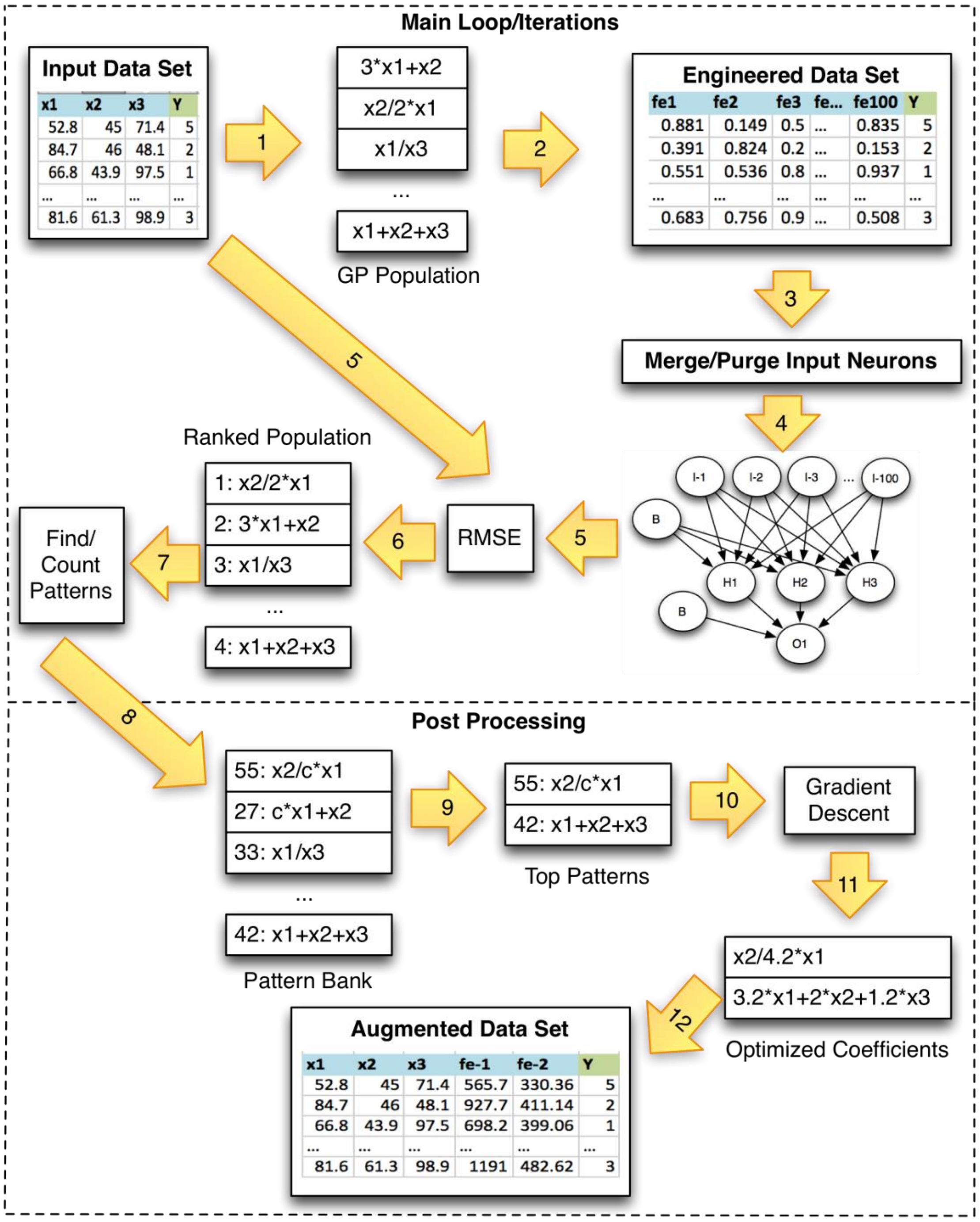


Figure 17. Overview of AFE algorithm

The numbered arrows in the above figure show the computer's progress through the algorithm. At the highest level, the algorithm is broken into two phases that are enclosed by dashed lines. The top box, which indicates the main loop/iterations is where most of the processing time is spent by the algorithm. The bottom box, which indicates the post processing steps, refine and aggregate the potential engineered features created by the first phase.

An input data set (before arrow 1) is provided to the algorithm. For this example, the input data set specifies a feature vector with three input variables: *x1*, *x2* and *x3*. A genetic programming population is created that contains expressions built of mathematical operators/functions, constants and these variables (arrow 1). This genetic programming algorithm evolves this population each iteration to provide patterns for the eventual set of engineered features. For this example, the population is of size 100. This population of expressions is used to generate an intermediate engineered feature data set that has 100 input variables that correspond to these expressions and the same expected values (*y*) as the original data set (arrow 2).

These input variables are standardized to z-scores and then mapped to the same 100 input variables of a deep neural network that is used to evaluate these 100 expressions in the population (arrow 3). The neural network shown in the above figure contains only a single hidden layer; however, for a real problem a neural network would be created per Appendix B, “Hyperparameters.” Any expressions that existed in the previous algorithm iteration were mapped to the same input neuron. The algorithm maps new expressions to input neurons whose expressions no longer exist in the population. These input neurons will have their weights reset to average values so that weights from ancestor expressions do not provide an invalid influence upon them. In this way, the neural network is constantly augmented from the previous generation and a new neural network is never built from scratch. The number of input neurons is always constrained to match the population size (100 for this example).

The neural network is trained to produce the same expected values (*y*) as the original data set until the neural network's weights stabilize (arrow 5). The RMSE of this neural network is tracked, and the input neurons are ranked per the perturbation ranking algorithm (arrow 6). These ranks become the scores for the corresponding 100 population members, thus filling the role of the objective function for the genetic program and completing an iteration for the AFE algorithm. At the end of each iteration, the top population members are scanned for common patterns (arrow 7). Patterns replace constants with a single token named *c*—thus patterns that differ only by their constants will not be considered discrete patterns. These constants were optimized in the post processing phase of the AFE algorithm. Iterations continue until the list of most common patterns does not change between iterations or some predefined maximum number of iterations occurs.

Once the iterations are complete, the pattern list is sent to the post processing phase (arrow 8). A configuration setting defines the number of the top patterns to optimize to become the final engineered feature set (arrow 9). Gradient descent optimizes the coefficient values of these top expressions to minimize the error of a neural network trained to produce the expected output (*y*) from the original data set (arrow 10). These optimized expressions become the set of engineered features produced by the AFE algorithm (arrow 11). An augmented data set is produced that contains the original data set inputs (*x*), the augmented features (*fe*), and the original data set's expected outcome (*y*). This augmented data set can be used to train any neural network framework, such as Encog or TensorFlow (arrow 12).

Feature Evaluation and Stochastic Scoring

Neural networks use stochastic gradient descent to decrease both training time and overfitting. Mini-batches are employed for each training iteration rather than the entire data set. These mini-batches allow neural networks to be trained against large data sets. This stochastic sampling technique can also sample the data sets of genetic programming algorithms (Tu & Lu, 2004). The AFE algorithm trains with stochastic batches for both neural network and genetic programming fitting. For more information on which hyperparameters were used in the experiments, refer to Appendix B, “Hyperparameters.”

The perturbation feature ranking algorithm (Breiman, 2001) was selected as the core feature ranking technology for the AFE algorithm. By ranking features, the entire population can be evaluated simultaneously. The relative positions of the candidate engineered feature in the feature rank becomes the score that the genetic selection operators evaluate when choosing the most fit candidate engineered feature (genome).

Neural Network Augmentation

A single neural network is trained through the entire run of the AFE algorithm to evaluate features. The single network prevents the need to retrain a neural network for each genetic programming step. However, the use of a common neural network requires some direct modification to the weights of the feature evaluation neural network as genomes are born and die over the genetic programming algorithm steps. When a new genome is born, it assumes some location in the feature vector of the evaluation neural network. The weights of that input neuron correspond to the old genome/feature that used to occupy that spot. This former occupant did not have a high enough ranking to remain in the next genetic algorithm step and was thus removed. To simply put the new genome/feature into this spot would likely disadvantage it. If the previous owner had a low enough score to be removed, then its weights were likely low. The weights of the input neuron that is about to receive the new genome must be set to neutral values. Consider Figure 18 that shows the weights that will need adjustment.

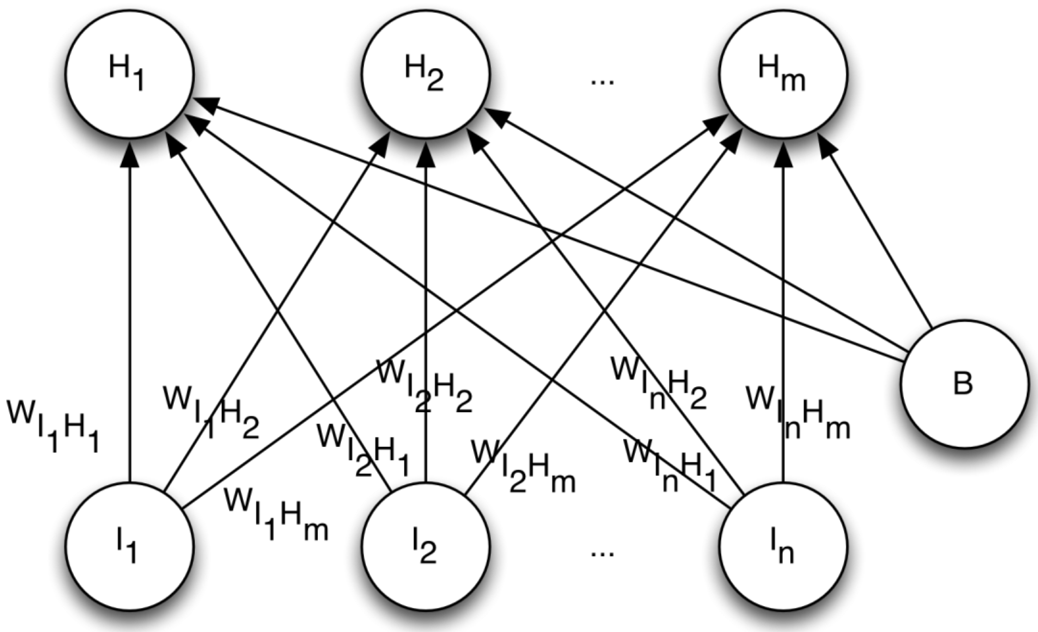


Figure 18. Input neuron weight connections

The new weights for an input neuron, which is about to be replaced, were set to the mean of the corresponding weights on all other input neurons. Bias neurons are not considered for this calculation. Each of a new input neuron’s weights can be determined from the following formula:

$$\boldsymbol{W_{I_j H_k} = \frac{\sum_{m \neq j} W_{I_m H_k}}{|I| - 1}} \tag{18}$$

The variable W represents the weights between an input neuron (I) and first layer hidden neuron (H). The denominator specifies the number of input neurons minus 1. There would never be 1 or fewer input neurons, as the number of input neurons equals the genetic programming algorithm's population size, which has an established minimum of 10.

Parameters for the AFE Algorithm

This section describes the usage of the AFE algorithm that was created for this dissertation research. The input to the AFE algorithm is a CSV file that specifies the inputs to a neural network (feature vector) and the expected output. The input/output values are assumed to be completely numeric. Any non-numeric columns, such as categorical values, should be encoded during preprocessing. It is assumed that all preprocessing has been carried out prior to running the AFE algorithm.

There are several parameters that are used to specify the operation of the AFE algorithm:

- Number of Engineered Features Desired (5) – The number of engineered features desired from the algorithm.
- Population Size (100) – How many candidate solutions are in the genetic programming population. A minimum population size of 10 is required.
- Neural Mini-Batch Size (32) – The size of each stochastic training batch for neural network training.

- GP Batch Size (32) – The size of each stochastic training batch for genetic programming.
- Evaluation Steps (5) – Evaluate feature importance after this number of steps.
- Ranking Stability Threshold (0.32) – When the Euclidian distance between two feature rankings is below this threshold, consider the ranking stabilized.

Default values are shown in parenthesis. Other hyperparameters for the neural network and genetic programming algorithms were set the same as they were for the experiments. Refer to Appendix B, “Hyperparameters,” for more details.

The algorithm begins by initializing a random population of expressions that form the initial generation of features. The genetic algorithm has a feature vector size equal to the feature vector size of the original data set. The neural network has a feature vector size equal to the population size of the genetic program. The flow of data through the feature vectors of the genetic programming algorithm and neural network is summarized in Figure 19.

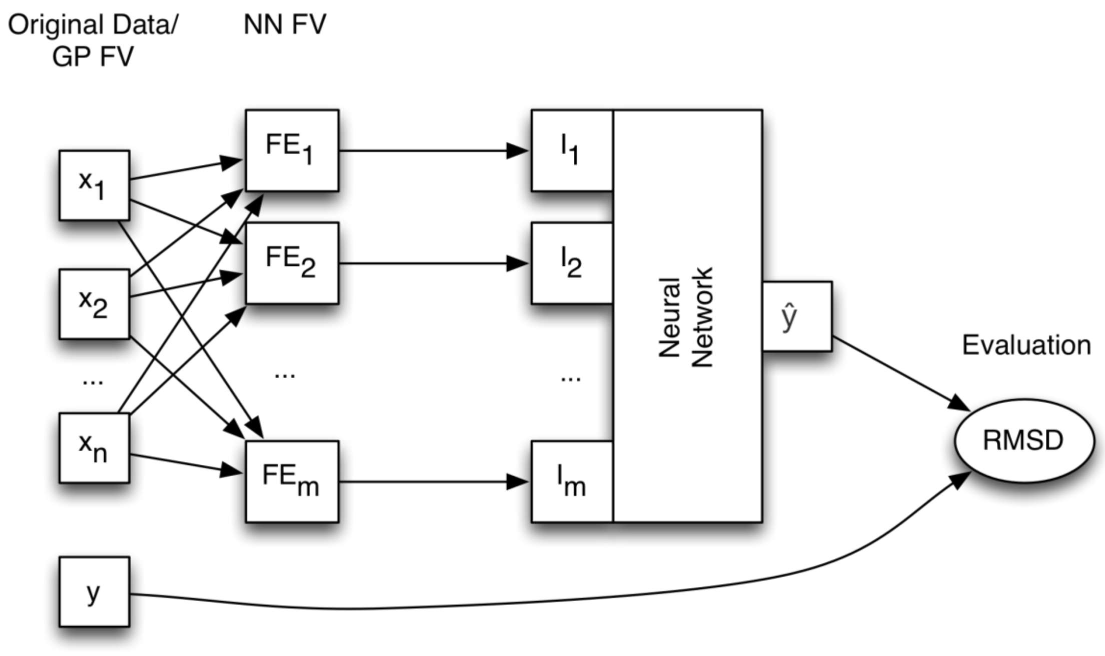


Figure 19. Data flow/mapping for the AFE algorithm

As the above figure illustrates, there are two feature vectors. The first feature vector comes from the original data set and feeds directly into the individual genetic programming expressions that become the engineered features. The outputs from each of these engineered features creates a second feature vector that is used to train the neural network.

AFE Algorithm Iteration

The AFE algorithm is iterative. Because an iteration only deals with a batch of the data, these iterations are steps (as opposed to epochs). Each step of the AFE algorithm is summarized by Figure 20.

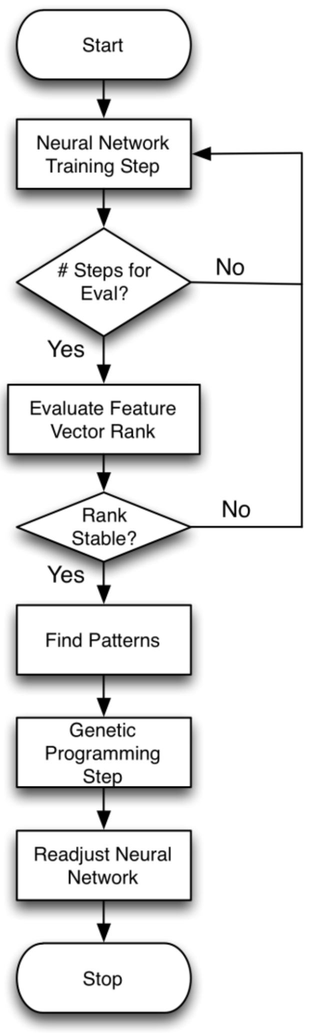


Figure 20. One step of the AFE algorithm

An iteration begins by training the neural network for one step. If the number of neural training steps has exceeded the evaluation steps parameter, then the feature rank is evaluated. The Euclidean distance is taken between this feature rank and the previous one. If this Euclidean distance is not below the ranking stability threshold parameter, the iteration ends.

If the rank stabilizes, then the top genomes in the population were evaluated for patterns and a genetic programming step was run. Common patterns accumulated over iterations and contributed the final set of engineered features. The genetic programming step will perform genome selection and crossover/mutation and produce the next generation of features. The neural network was adjusted for this next generation of features. These features/genomes were sorted so that genomes that survived to the next step remained in the same location—and thus retained their weights. For new features being added, it was important that their weights be initialized to new random values that were in a range consistent with the current training level of the neural network. The process for this initialization was covered earlier in the section, "Neural Network Augmentation."

Finding Patterns

Because genetic programs usually return a variety of solutions, often representing different local optima, it is important that the features be aggregated together from many runs. To accomplish this objective, the top ranked genomes were combined when they have similar form. This process is like the algorithm that was used to find patterns in Experiment 4. However, unlike Experiment 4, the constant values were averaged together, rather than eliminated. Averaging the constants together is not ideal; however, gradient descent later optimizes the constant averages in the final step of the AFE algorithm. Figure 21 shows how several features are combined.

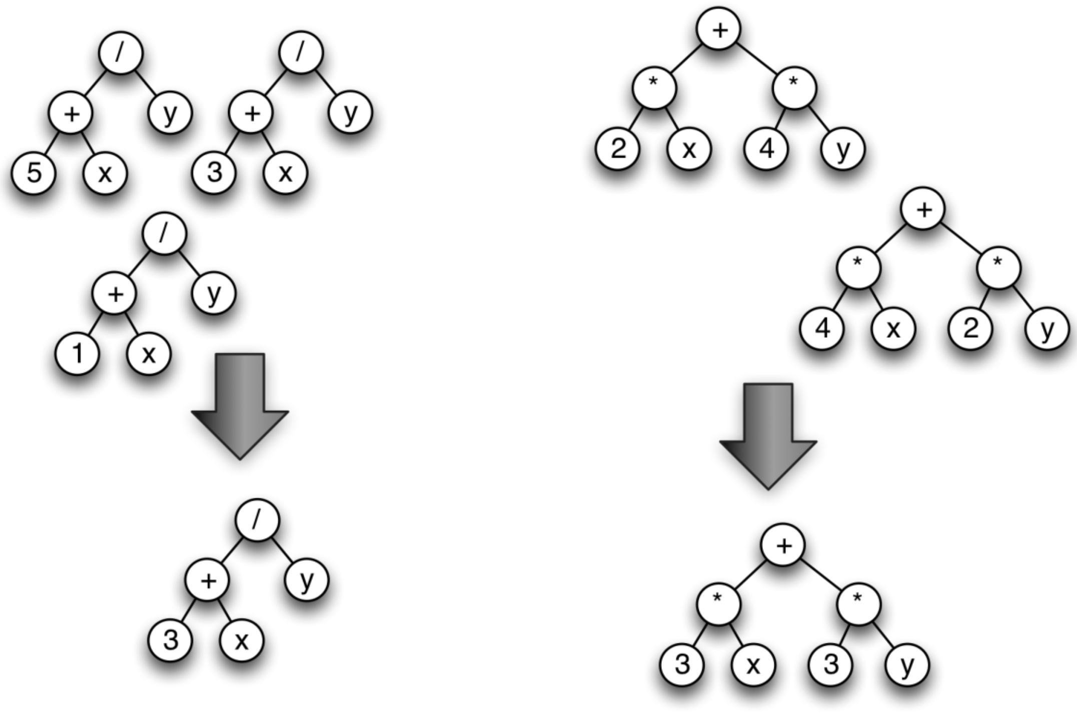


Figure 21. Collapsing trees

In Figure 21, the structure of the trees is maintained; however, the constants are combined by averaging. The algorithm selects the tree patterns that occur the most often as finalists to proceed to the last part of the algorithm. This aggregation also ensures that the final set of engineered features will have dissimilar structure—because common structures were aggregated together.

Optimizing the Final Set of Engineered Features

Once the patterns stabilize and the final set of engineered features can be defined, the coefficients of these engineered features can be optimized with gradient descent. This process is repeated for each of the set of final features. The constant coefficients of a genetic program can be optimized using gradient descent in the same way as the weights of a neural network (Topchy & Punch, 2001). This process can save the genetic search from a costly mutation-based search of the constant values.

Currently Encog does not support symbolic differentiation of genetic programs. Thus, finite difference is used to obtain derivatives of genetic programs. The following formula summarizes the finite differentiation process:

$$\boldsymbol{f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}} \tag{19}$$

The value *x* represents the variable for which the ordinary or partial derivative is being taken with respect to. The result of the limit approaches the derivative as the value of *h* approaches zero. The finite difference formula is used to take the partial derivative of each of the coefficients of the genetic program. Once the genetic programming algorithm converges to a stable solution, iterations of gradient descent are applied to further lower the objective function by finding more suitable values for the coefficients.

Experiment 6 Results

Experiment 6 evaluated the overall effectiveness of the AFE algorithm. Thus, Experiment 6 is the culmination of this research and demonstrates that the objectives in this dissertation have been met. Experiment 6 measured the amount of accuracy that was gained for neural networks with certain data sets. To ensure that this result was statistically significant, 100 experiment cycles were performed for each data set both with and without engineered features. In total, 200 runs were completed per data set. Finally, a Student's t-test demonstrated that the change in mean accuracy between the augmented and non-augmented data sets was statistically significant.

To perform this evaluation, the process of Experiment 2 was repeated with each data set being augmented with up to five engineered features from the algorithm. If the feature engineering algorithm was effective for a data set, adding the augmented features decreased the error for that data set. For example, the AFE algorithm suggested the following five engineered features for the wine data set:

$$-\frac{1.0}{1+0.061a}\left(7.54^{tp^{2.96}}-ah+\frac{alh}{9}+mg+(-1.88ci-f+5)^{-0.01}\right)^{\frac{ah}{h}} \tag{20}$$

$$-\frac{1}{0.06a-ah+f}\left(-0.08\left(\frac{1.67}{alh}\right)^{\left(f^{ma+2.71}\right)\frac{1}{aod}(16.44a-6.4od)-1}\left(ci^{tp}+72.99\right)\right)^{tp-8} \tag{21}$$

$$\frac{\left(7.54^{tp^{2.96}}-ah+\frac{10alh}{9}+mg-6\right)^{\frac{ah}{h}}}{1+0.06mg-0.01p} \tag{22}$$

$$-27^{\left(-\frac{1}{p-tp}(-0.1p+0.1tp+0.29)\right)(ci-7)}-6.94\cdot 10^{-171} \tag{23}$$

$$\frac{1}{1.55pr-11088.2}\left(18.28^{f}(pr-3)-4.68\cdot 10^{-11}pr+3.34\cdot 10^{-7}\right) \tag{24}$$

The variables in the above equations are defined as follows:

- *a* - alcohol
- *ah* - ash
- *alh* - alcalinity_ash
- *ci* - color_intensity
- *f* - flavanoids
- *h* - hue
- *ma* - malic_acid
- *mg* - magnesium
- *nfp* - nonflavanoid_phenols

- *od* - od28_od315
- *p* - proanthocyanins
- *pr* - proline
- *tp* - total_phenols

The genetic programming algorithm used by Encog preforms minimal algebraic simplification. The above equations were simplified and formatted using the Python SymPy algebraic system. Additionally, the constants above were truncated to two decimal places for readability. For a complete list of engineered features, across all data sets, refer to Appendix F, "Engineered Features for the Data Sets."

The AFE algorithm will always suggest the number of features that it was configured to return. The default is five features. These features are not guaranteed to improve neural network accuracy. The returned features are the ones that were measured to be the most important to neural network training. To demonstrate if these features cause greater accuracy, the AFE algorithm was compared with the baseline RMSE values from Experiment 2.

Table 13 shows the baseline neural mean accuracy and the mean accuracy with five automatically engineered features added.

Table 13. Experiment 6, feature engineered vs non-augmented accuracy

#	Name	Neural Mean	Auto Mean	Difference	Change Percent
6-1	Abalone	2.6523	2.8611	0.2087	0.07872
6-2	auto-mpg	3.3376	3.2369	-0.100	-0.03016
6-3	Bupa	0.4551	0.4490	-0.006	-0.01357
6-4	Covtype	0.3999	0.4140	0.0141	0.035392
6-5	Crx	0.3592	0.3428	-0.0164	-0.04575
6-6	Forestfires	0.0607	0.0619	0.0011	0.019726
6-7	Glass	0.3376	0.3306	-0.007	-0.02078
6-8	Heart	0.3377	0.3480	0.0102	0.03038
6-9	Hepatitis	0.4576	0.4392	-0.018	-0.0401
6-10	Horse	0.3659	0.3792	0.0133	0.0363
6-11	Housing	0.0446	0.0446	0	-0.0009
6-12	Iris	0.3182	0.3345	0.0162	0.05118
6-13	pima	0.4163	0.4212	0.0048	0.01176
6-14	soybean_large	0.1936	0.1779	-0.0157	-0.0811
6-15	wcbreast_wdbc	0.0988	0.1014	0.0025	0.02587
6-16	wcbreast_wpbc	0.3811	0.3752	-0.0059	-0.0156
6-17	wine	0.2520	0.2036	-0.0484	-0.1922

The first and second columns identify the experiment and data set. The third column shows the RMSE for a neural network with a regular data set (Experiment 2). The fourth column provides the results of data sets augmented with engineered features. Values shown in the fifth column give the amount that the error increased with the addition of the engineered features—a negative value is considered a good result. The sixth column summarizes this change as a percentage—likewise, a negative value is considered a good result. As this table demonstrates, several of the data sets benefited from feature engineering (FE). These results are shown graphically as Figure 22.

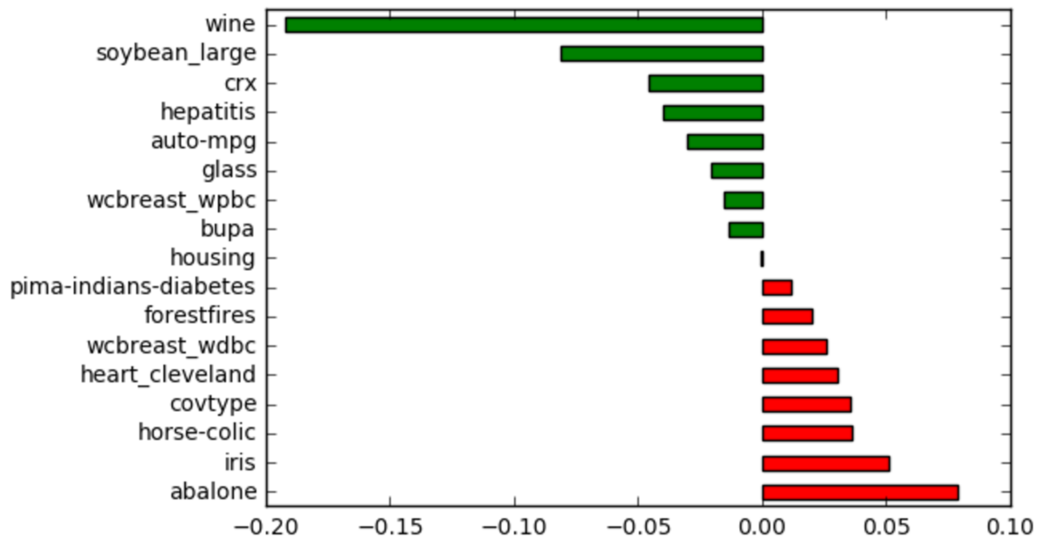


Figure 22. Experiment 6, feature engineering benefits to data set

Green bars in the above chart indicate decreases in the RMSE for these data sets:

- Wine
- Soybean_Large
- CRX
- Hepatitis
- Auto-MPG
- Glass
- WCBreast-WPBC
- BUPA

While these data sets showed a decrease in error (increase in accuracy), it is important to determine if this change is statistically significant. To make this determination, a Student's T-Test was performed between the experiment cycle results for the Experiment 2 results and the neural network accuracy when data sets were augmented with engineered features in Experiment 6. Table 14 shows the results of this T-Test for data sets where the percent increase in error was negative.

Table 14. Experiment 6, T-Test with data sets that showed improvement

#	Name	T-Statistic	P-Value
6-2	auto-mpg	0.38309	0.702064
6-3	bupa	1.190632	0.235223
6-5	crx	4.115739	**0**
6-7	glass	1.857998	0.064654
6-9	hepatitis	6.679419	**0**
6-11	housing	0.088687	0.92942
6-14	soybean_large	3.038647	**0.002697**
6-16	wcbreast_wpbc	1.125605	0.261695
6-17	wine	3.030937	**0.002764**

The null hypothesis is that the engineered features made no difference in the accuracy of the neural network. For any row in Table 14 where the p-value is below 0.05, the null hypothesis can be rejected. These numbers are bolded. Thus, the AFE algorithm produced a statistically significant increase in accuracy for the following data sets:

- crx
- hepatitis
- soybean_large
- wine

As previously stated, there was no expectation that the AFE algorithm would increase the accuracy for all data sets. Because the AFE algorithm caused a statistically significant increased accuracy of four data sets, the objective of this dissertation was met.

Summary

This chapter provided the results of all six experiments conducted for this dissertation. The first five experiments provided guidance for the creation of the AFE algorithm. Experiment 1 looked at which types of engineered features might benefit a neural network. Experiment 2 provided a baseline to compare data set results. Experiment 3 showed that neural networks and genetic programs can function together as ensembles. Experiment 4 analyzed many genetic programs to find common patterns for individual data sets. Experiment 5 provided results to determine the most effective means of ranking the relative importance of features in a neural network's feature vector.

The information from the five experiments enabled the creation of the AFE algorithm. This algorithm used a genetic algorithm to evolve candidate engineered features. A single neural network evaluated the entire population of potential features at once. As the neural network was trained, unfit features died, fit features survived, and new features were introduced. An analysis of top features from the population produced the final feature set. Finally, Experiment 6 was run to evaluate the effectiveness of the algorithm. Experiment 6 compared the results of data sets augmented with engineered features to the baselines established in both the second and third experiments. These results demonstrated that the AFE algorithm increased the neural network accuracy of four data sets by a statistically significant amount, fulfilling the objective of this dissertation.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

Conclusions

Neural networks will often benefit from feature engineering, which is the process of augmenting the feature vector input to a neural network with calculated expressions that are based on the other values in the feature vector. Genetic programming is a machine learning technique that builds mathematical expressions and computer programs to achieve an optimal score for an objective function. It was demonstrated in this dissertation that genetic programming can engineer features that increase the accuracy of neural networks. Feature engineering is not always effective for every data set, as was shown in this research. However, for some of the data sets evaluated in this research, the accuracy of a neural network was improved by a statistically significant amount with features engineered by the AFE algorithm. Thus, the goal of this research was achieved.

The AFE algorithm developed for this dissertation focused primarily upon relatively lower-dimension predictive analytics where the feature vector is made up of named input columns. Higher dimension data sets, such as computer vision, audio recognition, and time series were outside the scope of this dissertation. This research also focused on feedforward deep neural networks. Other neural network architectures, such as convolution neural networks (CNNs) and long short term memory (LSTMs) were likewise outside the scope of the research conducted for this dissertation.

Feature engineering research for lower-dimension predictive modeling using named fields (Anscombe & Tukey, 1963; Breiman & Friedman, 1985; Mosteller & Tukey, 1977; Neshatian, 2010; Tukey et al., 1982) and feature engineering for high dimension images, audio streams, and time series (Blei et al., 2003; M. Brown & Lowe, 2003; Coates et al., 2011; Coates & Ng, 2012; Le, 2013; Lowe, 1999; Scott & Matwin, 1999) have progressed along considerably different paths. Prior research into feature engineering for high dimension data has often involved dimension reduction or the extraction of features from the pixels of images. However, feature extraction from data sets with lower dimension named features often revolve around expressions among several input features. For this reason, this dissertation narrowed its scope to named fields for predictive modeling.

Data sets with many rows caused early versions of the AFE algorithm to run unacceptably slow. Through some experimentation and research, it was determined that a stochastic sampling of the training data for the genetic objective function yielded two advantages. First, not processing the entire data set required less runtime to handle the same data set. Second, not processing the entire data set helped decrease overfitting by sometimes preventing the genetic programming algorithm from entering local optima. Stochastic sampling of mini-batches is a common neural network technique and has yielded the same benefits to genetic programming. The algorithm developed for this dissertation found stochastic batches to be of value to both neural networks and genetic programming algorithms. Even though the AFE algorithm was implemented with the Encog machine learning framework, features engineered by this algorithm could be used by any deep learning neural network framework.

Implications

This research contributes several new directions to genetic programming research for deep neural networks. Prior research into genetic programming-based feature engineering for predictive modeling did not specifically target the unique characteristics of deep neural networks (Neshatian, 2010). Recent research has focused primarily upon using deep neural networks to engineer features for themselves (Davis & Foo, 2016; Kanter & Veeramachaneni, 2015; Le, 2013). Simplified proxy objective functions were often used by prior research due to the high computational cost of using neural networks in the objective function. The AFE algorithm developed in this dissertation solved the computation time execution problem using feature ranking to evaluate the entire population simultaneously.

An additional difficulty to the use of genetic programming-based feature engineering is the tendency for genetic programming to fall into local optima and produce many different candidate solutions for a given objective function. Experiments 1 and 2 observed this issue. To remedy this problem, the output candidate features from the genetic programming algorithm were combined and reduced to form the set of engineered features that the algorithm returns. The constant coefficients from this final set of features were optimized using gradient descent.

Recommendations

Feature engineering has seen considerable interest in the machine learning community. The results of this dissertation open additional areas of research for the application of genetic programming to feature engineering for neural networks. The following list represents some potential areas of research (in no specific order):

- Symbolic algebraic simplification could eliminate duplicate features when searching for common patterns in candidate engineered features.

- Symbolic differentiation could allow faster and more accurate optimization of the coefficients in the final step of the AFE algorithm. This would likely produce faster and more accurate coefficient tuning than the finite difference technique currently employed by the AFE algorithm.
- Evaluate effectiveness of this algorithm for recurrent neural networks, such as long short term memory (LSTM) networks. Support of LSTM would require enhancement of the Encog machine learning framework, which does not currently support LSTM.
- Currently the AFE algorithm will return the requested number of engineered features. This complete set of engineered features may not improve the accuracy of the neural network. It would be valuable to further extend the algorithm to only return features that will increase the accuracy of a neural network.
- The AFE algorithm can be computationally intense. While the dissertation code makes every attempt to be scalable across threads (vertical scalability), no provision is currently made for execution on a grid of computers (horizontal scalability).
- The research conducted in this dissertation only used genetic programming to evolve expressions. Moving from expressions to small programs might enable the genetic program to evolve routines that can extract features for images and time series.
- Provide an automated means of setting several of the configuration parameters that are necessary for the AFE algorithm.

- This dissertation focused only on the Encog machine learning framework. Other machine learning frameworks, such as Tensor Flow, Theano and The Microsoft Cognitive Toolkit (CNTK) each have different implementation details and optimizations of deep learning training algorithms. It would be possible to engineer features that are optimized to a specific deep learning framework by using that framework's neural network implementation to evaluate candidate features in the genetic programming objective function. This could result in greater accuracy for the specific deep learning framework being evaluated.

Summary

Introduction

Raw data are often preprocessed to become the input, or feature vector, to a neural network. Feature engineering is an important preprocessing step that augments the feature vector of a neural network with calculated values that are designed to enhance the accuracy of the neural network's predictions. Research has shown that the accuracy of deep neural networks sometimes benefits from feature engineering. Expressions that combine one or more of the original features usually create these engineered features. The choice of the exact structure of an engineered feature is dependent on the type of machine learning model in use. For example, engineered features that increase the accuracy of a deep neural network might not have a similar effect on a support vector machine (SVM). Typically, feature engineering is a manual process. However, research interest exists around automating aspects of feature engineering. A genetic programming-based means of AFE was developed for this dissertation research.

Genetic programming and deep neural networks both accomplish similar tasks. They accept an input vector and produce either a numeric or categorical output. However, internally, each functions differently. Deep learning adjusts large matrices of weights and biases to produce the desired output. Genetic programming constructs computer programs and expressions to yield the desired output. Because genetic programming is capable of evolving programs and expressions from its input features, it is logical to make it the basis for a feature engineering algorithm.

The primary challenge of this research was to limit the large search space of expressions that combined the features of the data set. This challenge was overcome by defining the types of expressions that most benefitted a deep neural network, determining the importance of a feature and prioritizing the expressions to be searched by the algorithm. Parallel processing was also employed to allow the algorithm to utilize multiple cores on the host computer system.

Additionally, the algorithm was designed to include an efficient objective function to evaluate candidate-engineered features against each other. Such a function was based on fitting a deep neural network model with candidate-engineered features. This objective function was designed to be efficient so that it could determine (in minimal time) how effective one candidate engineered feature was compared to another. Genetic programming uses this type of objective function to decide the best genomes (candidate-engineered features) to form the next generation of genomes. These design goals drove the methodology of this research.

Methodology

Six experiments in total were conducted to create an AFE algorithm. The first five experiments provided data that assisted in the design of the AFE algorithm. Characteristics such as feature ranking, common patterns in genetic expressions and genetic ensembles were all evaluated by these experiments. The sixth experiment evaluated the ability of the AFE algorithm to increase the accuracy of deep neural networks by a statistically significant amount.

The first experiment provided information that helped to limit the search space for the genetic programming algorithm. Deep neural networks will not necessarily see increased accuracy because of all expression types. For an engineered feature to increase the accuracy of a neural network, two conditions must be satisfied. First, the engineered feature must be an expression type that the neural network cannot easily synthesize on its own. Second, the engineered feature must expose some interaction between two or more original features. For example, BMI exposes an interaction between height and weight. The synthetic data set used for this experiment was made up of random inputs and the results from several common expression types. A high error (low accuracy) for a given expression indicates that the neural network is not particularly adept at synthesizing that feature. The results of this experiment provided information to focus the search space on expression types that most benefit a deep neural network.

The second experiment established a baseline with a variety of real world data sets. This baseline establishes the accuracy that can be expected from a regular deep neural network that has not had its data set augmented with engineered features. Real world data sets from the University of California Irvine machine learning repository were used. Because not all data sets benefit from feature engineering, it is important to use many data sets. The results from the second experiment were compared with the final experiment to determine what effect engineered features had on the accuracy of the neural network for a given data set.

The third experiment looked at the accuracy increase that might be achieved by using a neural network to combine ensembles of genetic programs. Combining multiple models together can be an effective means of increasing accuracy. This experiment was conducted to see the degree to which neural networks and genetic programs work as an ensemble. The results from this experiment were also used with the sixth experiment to ensure that any engineered features are performing beyond what could have been achieved with a simple ensemble of neural networks and genetic programs.

The fourth experiment demonstrated that common patterns would emerge when many genetic programs were fit to the same data set. Genetic programming algorithms are prone to solutions falling into local optima. The results of both Experiment 1 and Experiment 2 empirically demonstrate this result. This experiment evaluated the effect of finding patterns over large numbers of genetic programs fitted to the same data set. If certain patterns occur more frequently, they might indicate useful relationships between the features.

The fifth experiment evaluated two feature ranking algorithms to look at stability as the neural network was trained by backpropagation. Genetic programming algorithms must evaluate genomes to determine which are the most fit. The genomes used in this dissertation were candidate engineered features. A design goal of the AFE algorithm was to use a neural network to evaluate the candidates. To perform this evaluation, the entire population of genomes became the input to a neural network and feature ranking algorithms are used to determine the order of fitness among the genomes. Two feature ranking algorithms were considered: perturbation and weight analysis. A goal of the fifth experiment was to see which ranking algorithm stabilized fastest as a neural network was trained. Earlier stabilization meant that part of the costly training process for the neural network could be skipped. Unfortunately, neither algorithm was stable. Thus, the feature engineering algorithm was designed to use a neural network that was continuously trained as evolved features were added and unfit features removed. This process allowed the ranking algorithm to be used without costly retrains of the neural network.

Finally, the sixth experiment evaluated the degree to which the AFE algorithm increased the accuracy of neural networks for several data sets. The same data sets from

Experiments 1 and 2 were augmented with engineered features from the algorithm that were created during this research. The best result from either Experiment 1 or 2 was compared against the result from the augmented data set. The exact same type of neural network was used to train both Experiments 2, 3 and 5. A data set was considered to have its neural network accuracy improved if the Experiment 5 result was more accurate than Experiments 2 and 3.

Results

Experiment 1 demonstrated that certain expression types are difficult for neural networks to synthesize, while others were relatively easy for neural networks to synthesize. While certain expressions were difficult for neural networks, they were often synthesized by genetic programming. This result was an early indication that genetic programming would be useful for feature engineering. The results of Experiment 1 indicate the types of expressions that should be searched early.

Experiment 2 established the baseline for the ultimate evaluation of the AFE algorithm that was conducted in Experiment 6. Real world data from the University of California Irvine's machine learning repository were used. Both neural network and genetic programming algorithms were fit to these data. In most cases, the neural networks outperformed the genetic programing algorithm. This result was expected.

Experiment 3 showed the effect of using neural networks together with genetic programming in an ensemble. The ensembles produced mixed results—for some data sets a simple ensemble of neural networks and genetic programs could increase accuracy over neural networks alone. The purpose of this experiment was to establish a second baseline for ensembles of neural networks and genetic programs. By showing that the AFE algorithm exceeded both baselines, it was demonstrated that the AFE algorithm was not just an ensemble technique.

Experiment 4 dealt with the fact that genetic programming algorithms will often return a variety of solutions to the same data set over multiple runs. This experiment demonstrated that despite this behavior, the solutions returned over many runs of the genetic programming algorithm do converge to patterns. This result was a guiding factor in the AFE algorithm's ability to aggregate many genetic programming genomes together as the neural network trains.

Experiment 5 evaluated two feature ranking algorithms to see which was the most stable as the neural network was trained. This experiment concluded that neither the perturbation nor weight analysis algorithms were particularly stable. Thus, the perturbation algorithm was chosen because it considers the entire neural network, rather than simply the weights of the input neurons. Weight analysis algorithms consider only the input neuron weights. Perturbation has the advantage of analyzing the output of a neural network, thus all layers of the deep neural network contributed to the final feature ranking. It was also concluded from this experiment that features should be evaluated from a continuous run of a neural network, rather than individual runs for each genetic programming epoch.

The results of the first five experiments were considered to produce the AFE algorithm. This algorithm works by feeding the data set input to a genetic programming algorithm. A new data set is generated from the outputs of each of the population members and is used to train a neural network to produce the expected output from the original data set. Perturbation feature ranking is performed after a specified number of training steps and this ranking is used for selection during a genetic programming step. After the genetic programming step, top features/genomes were scanned for common patterns. A growing dictionary was kept of common patterns. The penultimate set of engineered features from the algorithm are the most common in this dictionary. The coefficients of these features are optimized with gradient descent and become the ultimate set that the algorithm returns.

The results of the sixth experiment demonstrated that data sets that are augmented with engineered features from the algorithm often achieve greater accuracy. Of the 18 data sets evaluated, four of them achieved statistically significant greater accuracy with the AFE algorithm. In none of these cases did an ensemble (Experiment 3) achieve more accuracy than the augmented features. In this respect, the objective of this dissertation was achieved.

Future Work

One area of future work is to improve the genetic programming algorithm so that it does not fall into local optima as easily. Ideally, the genetic programming algorithm should be able to consistently achieve a RMSE of 0 for all data sets in the first experiment. Decreasing overfitting and avoidance of local optima has been an area of research for genetic programming algorithms (Gao & Hu, 2006; Rocha & Neves, 1999; Spector & Klein, 2006). Such an enhancement to this algorithm would allow much quicker convergence to a set of engineered features. Additionally, a larger search space would be possible, perhaps allowing engineered features to be found that allow even greater accuracy.

Another area of future work is to integrate symbolic differentiation and algebraic simplification. Symbolic differentiation would allow faster, and more accurate, gradient descent than the current finite difference based differentiation. Current frameworks that support differentiation for Java should be investigated (Filice, Castellucci, Croce, & Basili, 2015). Additionally, more advanced algebraic simplification could allow duplicate patterns to be eliminated. This duplicate elimination would allow more robust location of patterns and a simplified form of the final engineered features. Current frameworks that provide algebraic simplification should be investigated (Albrecht, Buchberger, Collins, & Loos, 2012; Pinkus & Winitzki, 2002).

Appendixes

Appendix A
Algorithms

This appendix provides an overview of the artificial intelligence algorithms that were used to implement the AFE algorithm that was developed for this dissertation. The AFE algorithm was built upon standard algorithms—without substantial modification to them. Relatively minor differences exist in the implementations of many artificial intelligence algorithms. This appendix describes the assumptions on the algorithms used and their sources.

Feedforward Neural Network

A neural network accepts a fixed length input vector that is of the same length as the number of input neurons to the neural network. Typically, a synthetic value of 1 is appended to the end of the input feature vector to provide for a bias neuron. This bias neuron is essentially a y-intercept and allows the neural network to recognize patterns when all the other inputs are 0. The input layer will be connected to either the first hidden layer or directly to the output layer. Neural networks with no hidden layers are rare and ineffective for all but the simplest problems. The layer type dictates how the input layer is connected to the first hidden layer. There are many different layer types, such as dense, convolutional, dropout, maxpooling, contextual and spiking. This dissertation constrained its scope to only the dense layer type.

A dense layer contains connections from all input neurons (including the bias) to all neurons in the next layer (excluding that layer's bias). Bias neurons have a fixed output of 1 and do not have input connections. Because a bias neuron always outputs 1, it would not make sense to have input connections. All connections in a feed forward neural network are weighted. The value of a first hidden layer neuron is calculated by summing the weighted connections from all input layer neurons to that hidden neuron. A transfer function is applied to this summation before assigning these values to each of the first hidden layer neurons. The most common type of transfer functions for neural networks are the linear, softmax, sigmoidal, hyperbolic tangent and rectified linear unit (ReLU). The software provided for the AFE algorithm supports these transfer functions. However, the results demonstrated by this dissertation made the following assumptions: ReLU (Glorot et al., 2011) was used for hidden layers, linear was used for the output layer of regression networks and softmax was used for the output layer of classification neural networks.

This process is repeated for all layers of the neural network. The neuron values for the second hidden layer (including the bias value of 1) became the input to the third hidden layer or the output layer. Each layer of densely connected neurons are calculated from the preceding layer, all the way back to the input neurons. Once the output layer is calculated, the process stops. The values of the output neurons become the output for the neural network. The output layer does not have a bias neuron.

The process described in this section is the standard implementation of a feedforward neural network. This is the implementation used by Encog, TensorFlow and many others. For more information on basic neural network calculation refer to Goodfellow, Bengio, and Courville (2016).

Xavier Weight Initialization

Neural networks start with randomly initialized weights. Considerable research has been expended into the initialization of weights that converge quickly to a solution.

Different sets of random starting weights can produce neural networks of varying levels of success. Any such variance in the success of an algorithm can make empirical measurement difficult. Because of this it was important to find a weight initialization algorithm that produced the most consistent results available for feedforward neural networks. For this reason, the Xavier weight initialization algorithm performed the initialization of all neural networks in this dissertation research.

The Xavier weight initialization algorithm has been demonstrated to provide weights that converge to good solutions with minimal variance (Glorot & Bengio, 2010). This algorithm initializes the weights between layers of a neural network with a uniform random number distribution and a variance per the following equation.

$$\mathbf{Var}(\boldsymbol{W}) = \frac{\mathbf{2}}{\boldsymbol{n_{in}} + \boldsymbol{n_{out}}} \tag{25}$$

The variable W represents the weights between two layers where the input layer neuron count is represented by n_{in} and the output layer is represented by n_{out}. Bias neurons are counted for the input layer but not the output layer. Weight matrixes are always sandwiched between two layers—an input and output (relative to the weight matrix).

Mini Batch Stochastic Gradient Descent

Backpropagation is a very common technique for neural network training. This technique works by taking the partial derivative of the error function of the neural network with respect to each weight, with all other weights held constant. Each weight's individual partial derivative is evaluated and the weight is modified per the update rule in use. There are several different update rules that can be used with backpropagation. This dissertation performed all training using the Adaptive Moment Estimation (Adam) update rule that is described in the next section.

This dissertation makes use of backpropagation with stochastic mini-batches, using the Adam update rule to perform gradient descent. The term mini-batch implies that the neural network weights are updated in relatively small batches. A typical mini-batch size is under 100 training set elements. The partial derivatives of the weights, with respect to the error function, are evaluated and summed. These summations of one or more partial derivatives are called the gradients. There is one gradient for every weight and bias weight in the neural network. Once the gradients are obtained, the update rule specifies how the weights are to be modified. For more information on the mini-batch size and parameters used by this dissertation, refer to Appendix B, "Hyperparameters."

Adaptive Moment Estimation (Adam) Update Rule

The traditional momentum-based backpropagation update rule requires that learning rate and momentum parameters be set to values to govern the training. The learning rate specifies how much of a gradient should be applied to the corresponding weight. The momentum parameter specifies how much of the previous step's weight delta should be applied to the current step. As its name implies, momentum causes the direction of weight modification to be more difficult to change and has been demonstrated to help the neural network weights move through local optima.

There are two primary problems with the traditional approach to learning rate and momentum. The first issue is that the same learning rate and momentum must be applied to all weights. While there are algorithms that will change the learning rate and momentum as training progresses, what was needed was a means of allowing each weight to have its own learning rate. The literature review of this dissertation surveyed several approaches to individual learning rates. The chosen update rule is the Adaptive Moment Estimation (Adam). Adam uses several statistical measures to estimate a proper learning rate for each weight. Though Adam does have an initial learning rate as one of its training parameters, the algorithm will quickly adjust this to a more optimal rate for each weight. Additionally, Adam is considered (by its implementers) to be forgiving of non-optimal training parameter settings. This dissertation used a standard implementation of Adam, provided by Encog, with all training parameters set as recommended by the Adam seminal paper (Kingma & Ba, 2014).

Perturbation Feature Ranking

Feature ranking algorithms are critical to the AFE algorithm that was presented by this dissertation. The dissertation algorithm maps input neurons to the population of candidate engineered features. A feature ranking algorithm is used to determine which candidate engineered features are the most important to the neural network achieving a high accuracy rate. The feature ranking algorithm effectively becomes the objective function for the genetic algorithm code that evolves the candidate engineered feature population.

The feature ranking algorithm that was chosen for this dissertation is the perturbation feature ranking algorithm (Breiman, 2001). This algorithm is relatively simple, but effective (Olden et al., 2004). To evaluate the importance of the individual features the data set is evaluated once with each feature perturbed and all other features left alone. Features are perturbed by shuffling their contents. This causes that feature to be essentially useless for prediction, yet the column still retains the same statistical properties of minimum, maximum, mean, and standard deviation. The perturbation algorithm is useful because a new neural network does not need to be trained for each of the features. The feature being evaluated remains in the neural network, changes are only made to data.

The feature perturbation algorithm used by the AFE algorithm was provided by the Encog framework. Encog's implementation of perturbation feature ranking is based on the work of Breiman (2001) and Olden et al. (2004).

Genetic Programming

Genetic programming is not as popular of a technique as deep learning. Because of this there are many different implementations of it that follow a variety of different standards. The genetic programming algorithm that this dissertation used was provided by the Encog framework. Encog implements tree based genetic programming that is based on the work of Poli et al. (2008), which is based on the seminal work of Koza (1992).

The algorithm makes use of tournament selection to choose optimal parents. Trees are expressed internally as Java object structures but can be imported from infix or reverse polish notation (RPN). Encog can export these trees to infix, RPN or LaTex. The initial population is created with the ramped half and half algorithm (Koza, 1992). The dissertation algorithm uses threshold speciation, provided by Encog, that is based on research done by Stanley and Miikkulainen (2002).

Appendix B

Hyperparameters

Hyperparameters are settings that govern the operation of machine learning models. For a neural network hyperparameters include the number of hidden layers and the choice of transfer function. Training parameters, such as learning rate and momentum are also considered. Hyperparmeter selection can have a considerable impact on the performance of a machine learning model, such as a neural network or genetic programming algorithm. This appendix describes the hyperparameters that were chosen for this dissertation.

Main Hyperparameters Configuration

Most hyperparameters used by the AFE algorithm and the experiment runner framework are defined in the following Java source file:

```
   com.jeffheaton.dissertation.JeffDissertation
```

The settings that were used to generate the findings reported in the dissertation are given in this appendix. Because the dissertation source code is contained on a GitHub server, the author may have made improvements to these settings for increased performance. Because such improvements might be made after the publication of this dissertation, the original set is provided here. Additionally, a Git tag captures the final source code for the publication of this dissertation. For more information on the dissertation source code, refer to Appendix H, “Dissertation Source Code.”

```
/**
 * How many times should neural network experiments be
 * repeated.  The mean, min, max and standard deviation of
 * these runs are reported.
 */
```

```
public static final int NEURAL_REPEAT_COUNT = 100;

/**
 * How many times should genetic programming experiments be
 * repeated.  The mean, min, max and standard deviation of
 * these runs are reported.
 */
public static final int GENETIC_REPEAT_COUNT = 100;

/**
 * The random seed that is used for test/training splits.
 * Note that neural networks and genetic programs are
 * initialized without a seed (seed is based on current
time).
 * This ensures that each repeated run is different
weights.
 */
   public static final long RANDOM_SEED = 42;

/**
 * The starting learning rate for the Adam update.
 */
public static final double LEARNING_RATE = 1e-2;

/**
 * How many steps before a neural network is considered
 * stagnant and early stopping should apply.
 */
public static final int STAGNANT_NEURAL = 100;

/**
 * How many steps before a genetic program is considered
 * stagnant and early stopping should apply.
 */
public static final int STAGNANT_GENETIC = 100;

/**
 * L1 regularization for neural network training.
 */
public static final double L1 = 0;

/**
 * L2 regularization for neural network training.
 */
public static final double L2 = 1e-8;

/**
```

```
 * The percent of data to use for training.  Validation
data
 * are used for early stopping.
 */
   public static final double TRAIN_VALIDATION_SPLIT =
0.75;

/**
 * The mini batch size for stochastic gradient descent.
 */
public static final int MINI_BATCH_SIZE = 32;

/**
 * The population size to use for genetic programming.
 */
public static final int POPULATION_SIZE = 100;

/**
 * The update rule to use.
 */
public static final Class UPDATE_RULE = AdamUpdate.class;

/**
 * The minimum improvement amount to consider for early
stopping.
 */
public static final double MINIMUM_IMPROVE = 0.01;

/**
 * The number of GP runs to use to look for patterns in
experiment 4.
 */
public static final int EXP4_PATTERN_COUNT = 10;/** **/
```

Neural Network Architectures

A neural network's architecture has a great impact on its performance. For transfer functions, this dissertation followed the advice of Goodfellow et al. (2016), Glorot et al. (2011), and Bottou (2012). Hidden layers made use of a rectifier linear unit (ReLU). For classification data sets a softmax transfer function was used on the output layer. For regression data sets a linear transfer function was used on the output layer. Xavier weight initialization was used to set the weights to initial random values. Data were divided into training and validation sets. The training set was made up of 75% of the original data set and the remaining 25% became the validation set. These splits were from a Mersenne Twister random number generator using a seed of 42.

The first hidden layer of the neural network will contain twice the neurons of the input layer. Each subsequent hidden layer will contain half of the previous layer's number of neurons. The Encog neural network for all experiments and the AFE algorithm is created by the following code:

```
// Create the neural network and input layer
BasicNetwork network = new BasicNetwork();
network.addLayer(new BasicLayer(null, true, inputCount));

// First hidden layer is twice the number of inputs
int hiddenCount = inputCount * 2;

// Create the first hidden layer
network.addLayer(new BasicLayer(
  new ActivationReLU(), true, hiddenCount));

// Create the second hidden layer
hiddenCount = Math.max(2, hiddenCount/2);
network.addLayer(new BasicLayer(
  new ActivationReLU(), true, hiddenCount));

// Create the third hidden layer
hiddenCount = Math.max(2, hiddenCount/2);
network.addLayer(new BasicLayer(
  new ActivationReLU(), true, hiddenCount));

// Create the fourth hidden layer
hiddenCount = Math.max(2, hiddenCount/2);
network.addLayer(new BasicLayer(
  new ActivationReLU(), true, hiddenCount));

// Create the output layer
if (regression) {
  network.addLayer(new BasicLayer(
    new ActivationLinear(), false, outputCount));
} else {
  network.addLayer(new BasicLayer(
    new ActivationSoftMax(), false, outputCount));
}

// Finalize the network and randomize the weights
network.getStructure().finalizeStructure();
(new XaiverRandomizer()).randomize(network);
return network;
```

Neural Network Training Parameters

Neural network training parameters can have a substantial effect on the accuracy of the neural network. For this dissertation, training parameters were chosen to minimize the variance between experiment cycles. To accomplish this goal stochastic gradient descent, with the Adam update rule was used. An early stopping rule was setup using the 25% validation set. The code used to create the neural network trainer made use of the constants defined earlier in this appendix. The actual code used to create all neural network trainers in this dissertation is given here:

```
StochasticGradientDescent train =
    new StochasticGradientDescent(network, trainingSet);

train.setUpdateRule(JeffDissertation.factorUpdateRule());

train.setBatchSize(JeffDissertation.MINI_BATCH_SIZE);
train.setL1(JeffDissertation.L1);
train.setL2(JeffDissertation.L2);

train.setLearningRate(JeffDissertation.LEARNING_RATE);

EarlyStoppingStrategy earlyStop = null;

if( validationSet != null) {
  earlyStop = new EarlyStoppingStrategy(validationSet,
  5, JeffDissertation.STAGNANT_NEURAL);

  earlyStop.setSaveBest(true);
  earlyStop.setMinimumImprovement(
      JeffDissertation.MINIMUM_IMPROVE);
      train.addStrategy(earlyStop);
}

return new DissertationNeuralTraining(train,earlyStop);
```

Genetic Programming Training Parameters

This dissertation used the Encog genetic programming framework at several points. The first step to using the Encog genetic programming framework is to define the set of operators used. The input variables must also be defined in the Encog context. The following code provides this:

```
EncogProgramContext context = new EncogProgramContext();

// Define variables
for (String field: fields) {
    context.defineVariable(field);
}

// Define functions used
FunctionFactory factory = context.getFunctions();

factory.addExtension(
   StandardExtensions.EXTENSION_VAR_SUPPORT);
factory.addExtension(
   StandardExtensions.EXTENSION_CONST_SUPPORT);
factory.addExtension(
   StandardExtensions.EXTENSION_NEG);
factory.addExtension(
   StandardExtensions.EXTENSION_ADD);
factory.addExtension(
   StandardExtensions.EXTENSION_SUB);
factory.addExtension(
   StandardExtensions.EXTENSION_MUL);
factory.addExtension(
   StandardExtensions.EXTENSION_DIV);
factory.addExtension(
   StandardExtensions.EXTENSION_POWER);
```

Once the context is set up, the population can be created and the objective function defined. The trainer is created and a standard ratio of mutation and crossover is defined. Algebraic rewrite rules are added to simplify some expressions. Early stopping is defined, based on the validation set. A new random population is created using the ramped half and half initializer. Finally, the results are returned to whatever experiment is making use of the genetic programming algorithm. The following code listing demonstrates this process:

```
// Create the population
PrgPopulation pop = new PrgPopulation(
    context, populationSize);

// Create the objective function
MultiObjectiveFitness score = new MultiObjectiveFitness();
score.addObjective(1.0, new TrainingSetScore(trainingSet));

// Create the trainer
TrainEA genetic = new TrainEA(pop, score);
genetic.setCODEC(new PrgCODEC());
genetic.addOperation(0.5, new SubtreeCrossover());
genetic.addOperation(0.25, new ConstMutation(context, 0.5,
1.0));
genetic.addOperation(0.25, new SubtreeMutation(context,
4));
genetic.addScoreAdjuster(new ComplexityAdjustedScore(10,
20, 50, 100.0));

// Add algebra rewrite rules
pop.getRules().addRewriteRule(new RewriteConstants());
pop.getRules().addRewriteRule(new RewriteAlgebraic());
pop.getRules().addConstraintRule(new SimpleGPConstraint());
genetic.setSpeciation(new PrgSpeciation());
genetic.setThreadCount(1);
```

```
// Define early stopping
EarlyStoppingStrategy earlyStop = null;
if( validationSet != null ) {
    earlyStop = new EarlyStoppingStrategy(
      validationSet, 5, JeffDissertation.STAGNANT_GENETIC);
    earlyStop.setMinimumImprovement(
       JeffDissertation.MINIMUM_IMPROVE);
               genetic.addStrategy(earlyStop);
}

// Create new population
(new RampedHalfAndHalf(context, 1, 6)).generate(
   new Random(), pop);

genetic.setShouldIgnoreExceptions(false);

return new JeffDissertation.DissertationGeneticTraining(
    pop,earlyStop,genetic);
```

Appendix C

Third Party Libraries

This dissertation made use of several third-party libraries for the Python and Java programming languages. This appendix lists these libraries and provides the appropriate version and sources for these packages. All third-party libraries were used as they were provided—no source code changes were made. Any customization of these libraries was accomplished using configuration settings.

Java Third Party Libraries

Java 1.8[2] was the primary programming language used for this dissertation research. This section describes the Java-based third party libraries used in conjunction with Java.

Gradle

Gradle 3.2 was used to build the Java framework project that ran all the experiments for the dissertation research. Gradle can create a fat JAR that contains all the dependencies for the dissertation experiment framework. This allowed easy execution on the Amazon Elastic Cloud by uploading a single JAR file. For more information on how to obtain and run the experiment framework, refer to Appendix H, "Dissertation Source Code Availability."

Gradle allows dependencies to be specified using a special encoding string. This allows very precise specification of the version of third party libraries that were used. The Gradle dependency string is provided in the following sections for each of the third-party library packages discussed.

[2] Available from https://www.java.com/

Encog

Encog 3.4[3] is the machine learning framework that provided the deep learning and genetic programming components for this research. The author of this dissertation is a major contributor to the Encog project. The Gradle dependency string for the version of Encog used in this dissertation is 'org.encog:encog-core:3.4.0’.

OpenCSV

OpenCSV 2.3[4] provided all comma separated value file (CSV) reading and writing functions for this dissertation research. All data files needed for this dissertation are provided in CSV format. Additionally, output reports from the dissertation experiment framework were written to CSV format. The Gradle dependency string for the version of OpenCSV used is 'net.sf.opencsv:opencsv:2.3'.

JSON

The JSON library, version 20160212, provided the ability to read the configuration files for this dissertation research. The Gradle dependency string for the version of the JSON library that was used for this research is: 'org.json:json:20160212'.

JAMA

JAMA[5] 1.0.3 (Boisvert et al., 1998) is a software library for performing numerical linear algebra tasks. JAMA was created at National Institute of Standards and Technology in 1998 similar in functionality to LAPACK. JAMA provided some linear algebra support needed by these experiments. The Gradle dependency string for the version of JAMA that was used for this research is ‘gov.nist.math:jama:1.0.3'.

[3] Available from http://www.encog.org
[4] Available from http://opencsv.sourceforge.net/
[5] Available from http://math.nist.gov/javanumerics/jama/

Python Third Party Libraries

The Anaconda distribution of Python 3.5.2[6] produced the charts and table statistics for the experiment runs. Additionally, Python was used to occasionally validate the results of Encog to other machine learning frameworks.

TensorFlow

TensorFlow 0.8[7] was used to occasionally validate Encog deep learning network algorithms. Google released TensorFlow as an open source software library for machine learning.

Pandas

Pandas 0.19.1[8] provided the data set handling for the data obtained when the dissertation experiments were executed. By working directly with Scikit-Learn, NumPy, and Matplotlib, Pandas allows the experimental result data to be summarized and presented in its final form.

Scikit-Learn

Scikit-Learn 0.18[9] is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy. This dissertation used Scikit-Learn to interface to the SKFlow framework in TensorFlow.

---

[6] Available from https://www.continuum.io/downloads
[7] Available from https://www.tensorflow.org/
[8] Available from http://pandas.pydata.org/
[9] Available from http://scikit-learn.org/stable/

SciPy

SciPy 0.18.1[10] is an open source Python library used for scientific and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering. The statistical analysis of the results of this dissertation were calculated with SciPy.

Numpy

NumPy 1.11.1[11] is an extension to the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these structures. Aggregation of the many experiment cycles were performed using NumPy.

MatplotLib

Matplotlib 1.5.3[12] is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like wxPython, Qt, or GTK+. Charts used to present experiment results were produced using matplotlib.

[10] Available from https://www.scipy.org/
[11] Available from http://www.numpy.org/
[12] Available from http://matplotlib.org/

Appendix D

Data Set Descriptions

Several real-world and synthetic data sets allowed the evaluation of the AFE algorithm that was developed for this dissertation. The complete list of data sets is shown in Table 15:

Table 15. Data sets used by the dissertation experiments

Type	Name	Y	X	Experiments
r	abalone	rings		2,3,4,5,6
r	auto-mpg	mpg		2,3,4,5,6
r	bupa	drinks		2,3,4,5,6
c	covtype	cover_type		2,3,4,5,6
r	crx	a16		2,3,4,5,6
r	feature_eng	ratio_poly-y0	ratio_poly-x0	1
r	feature_eng	ratio_diff-y0	ratio_diff-x0, -x1, -x2, -x3	1
r	feature_eng	ratio_poly2-y0	ratio_poly2-x0, -x1	1
r	feature_eng	sqrt-y0	sqrt-x0	1
r	feature_eng	log-y0	log-x0	1
r	feature_eng	poly2-y0	poly2-x0, -x1	1
r	feature_eng	poly-y0	poly-x0	1
r	feature_eng	pow-y0	pow-x0	1
r	feature_eng	diff-y0	diff-x0, -x1	1
r	feature_eng	coef_ratio-y0	coef_ratio-x0, -x1	1
r	feature_eng	ratio-y0	ratio-x0, -x1	1
c	forestfires	area		2,3,4,5,6
c	glass	type		2,3,4,5,6
c	heart_cleveland	num		2,3,4,5,6
c	Hepatitis	class		2,3,4,5,6
c	horse-colic	outcome		2,3,4,5,6
c	housing	Crim		2,3,4,5,6
c	iris	Species		2,3,4,5,6
c	pima-indians	Class		2,3,4,5,6
c	soybean_large	Type		2,3,4,5,6
c	wcbreast_wdbc	Diagnosis		2,3,4,5,6
c	wcbreast_wpbc	Outcome		2,3,4,5,6
c	Wine	Class		2,3,4,5,6

The first column specifies if the data set is used for classification (c) or regression (r). The second column provides the name of the data set. The third column provides the target (or y) column for the data set. If all remaining columns are to be used as predictors, the fourth column is blank. If only a subset of the columns is to be used as predictors, they are listed in the fourth column. Finally, the fifth column lists the experiments that make use of each data set.

The synthetic data sets are combined into the data set named *feature_eng*. All remaining data sets are real world data sets from the UCI Machine Learning Repository (Newman & Merz, 1998). The synthetic data sets are described in greater detail in Appendix E, “Synthetic Data Set Generation.” The real-world data sets are described in the following sections.

Abalone Data Set (Abalone)[13]

Data from a study that provides physical measurements of abalone used to predict the age. The actual age of an abalone is determined by cutting the shell through the cone, staining it, and counting the number of rings through a microscope. Other measurements, which are easier to obtain, are used to predict the age. For this dissertation, a regression model was built between the rings (age) and the other predictors.

Auto MPG Data Set (Auto-mpg)[14]

Attributes that describe automobiles are used to predict the miles per gallon (MPG) for these automobiles. This dataset is a slightly modified version of the dataset provided in the StatLib library. Additionally, 8 of the original instances were removed because

[13] Available from: https://archive.ics.uci.edu/ml/datasets/Abalone
[14] Available from: https://archive.ics.uci.edu/ml/datasets/Auto+MPG

they had unknown values for the "mpg" attribute. For this dissertation, a regression model was built between the car's miles per gallon (MPG) rating and the other predictors.

Liver Disorders Data Set (Bupa)[15]

A data set that examines the correlation between the number of daily alcoholic beverages and 5 blood tests which are thought to be sensitive to liver disorders that might arise from excessive alcohol consumption. Each line in the data set constitutes the record of a single male individual. For this dissertation, a regression model was built between the number of beverages consumed and the other predictor columns.

Covertype Data Set (Covtype)[16]

This data set examines the correlation between forest cover type and cartographic variables. The actual forest cover type for a given observation (a 30 x 30-meter cell) was determined from US Forest Service (USFS) Region 2 Resource Information System (RIS) data. For this dissertation, a classification model was built between the cover type and the other predictor columns.

Credit Approval Data Set (Crx)[17]

This data set investigated predicting credit decision class based on several redacted fields from credit card applications. All attribute names and values have been changed to meaningless symbols to protect confidentiality of the data. This dataset is a good mix of attributes—continuous, nominal with small numbers of values, and nominal with larger numbers of values. There are also a few missing values. For this dissertation, a

[15] Available from: https://archive.ics.uci.edu/ml/datasets/Liver+Disorders
[16] Available from: https://archive.ics.uci.edu/ml/datasets/Covertype
[17] Available from: https://archive.ics.uci.edu/ml/datasets/Credit+Approval

classification model was built between the redacted *a16* field (class) and the other predictor columns.

Forest Fires Data Set (Forestfires)[18]

This data set examines the correlation between the number of burned acres in specific regions with other attributes of those regions. For this dissertation, a regression model was built between the area column and the other predictor columns.

Glass Identification Data Set (Glass)[19]

This data set investigated the correlation between the type of glass and refractive index and percentage of magnesium, aluminum, silicon, potassium, calcium, barium, and iron. For this dissertation, a classification model was built between the glass type column and the other predictor columns.

Heart Disease Data Set (Heart)[20]

This data set contains 76 attributes, however all published experiments refer to using a subset of 14 of them. This dissertation used Cleveland database, as is it the only one that has been used by ML researchers to this date. The *goal* column refers to the presence of heart disease in the patient. It is integer valued from 0 (no presence) to 4 (presence). Experiments with the Cleveland database have concentrated on simply attempting to distinguish presence (values 1,2,3,4) from absence (value 0). For this dissertation, a classification model was built between the goal column and the other predictor columns.

[18] Available from: https://archive.ics.uci.edu/ml/datasets/Forest+Fires
[19] Available from: https://archive.ics.uci.edu/ml/datasets/Glass+Identification
[20] Available from: https://archive.ics.uci.edu/ml/datasets/Heart+Disease

Hepatitis Data Set (Hepatitis)[21]

A data set that shows correlations between several medical tests/observations and the survival of patients with hepatitis. For this dissertation, a classification model was built between the patient's survival (live or die) and the other predictor columns.

Horse Colic Data Set (Horse)[22]

This data set looks at the outcome of horses that were treated for colic. The outcome was either: lived, died, or was euthanized. For this dissertation, a classification model was built between the treatment outcome and the other predictor columns.

Housing Data Set (Housing)[23]

This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University. For this dissertation, a classification model was built between the crime column and the other predictor columns.

Iris Data Set (iris)[24]

The classic Iris Data Set allows models to predict iris species from four measurements. This classic data set was included primarily for testing purposes. For this dissertation, a classification model was built between the iris species column and the other predictor columns.

[21] Available from: https://archive.ics.uci.edu/ml/datasets/Hepatitis
[22] Available from: https://archive.ics.uci.edu/ml/datasets/Horse+Colic
[23] Available from: https://archive.ics.uci.edu/ml/datasets/Housing
[24] Available from: http://archive.ics.uci.edu/ml/datasets/Iris

Pima Indians Diabetes Data Set (Pima)[25]

The results of a study to predict presence of diabetes in female patients that are at least 21 years old of Pima Indian heritage. For this dissertation, a classification model was built between the class (value of 1 indicates diabetes) column and the other predictor columns.

Soybean (Large) Data Set (Soybean)[26]

The Soybean Disease Diagnosis data set consisted of 19 disease classes. Only the first 15 of which have significant training samples. The predictor columns contain 35 categorical attributes, some nominal and some ordered. The values for attributes are encoded numerically, with the first value encoded as "0," the second as "1," and so forth. All unknown values were encoded as “?”. For this dissertation, a classification model was built between the disease type column and the other predictor columns.

Breast Cancer Wisconsin P & D (WcBreastP & WcBreastD)[27]

There are two breast cancer data sets provided to the University of California Irvine machine learning repository by the University of Wisconsin, Madison that are used by this dissertation. These two data sets are known as the Wisconsin Diagnostic Breast Cancer data set (WDBC) and the Wisconsin Prognostic Breast Cancer (WPBC) data set. Both use various predictors to determine the diagnosis and outcome respectively. Both are classification problems for this dissertation.

[25] Available from: https://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes
[26] Available from: https://archive.ics.uci.edu/ml/datasets/Soybean+(Large)
[27] Available from: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

Wine Data Set (Wine)[28]

The wine data set contains data that are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines, labeled class 1, 2 and 3. For this dissertation, a classification model was built between the class column and the other predictor columns.

[28] Available from: https://archive.ics.uci.edu/ml/datasets/Wine

Appendix E

Synthetic Data Set Generation

Experiment 1 made use of a synthetic data set that generates random samples from several equation types. This data set is used to test the ability of a neural network to synthesize several feature types that were shown in Table 8. This data set was generated with the following Python script:

```
import inspect

import numpy as np
import pandas as pd
import math

NUM_ROWS = 50000
SEED = 42
OUTLIER_THRESH = 2.5

NAME = 'name'
FN = 'fn'

GENERATED_FEATURES = [
    {NAME: 'ratio_diff', FN: lambda a, b, c, d: (a - b) /
(c - d)
     },
    {NAME: 'diff', FN: lambda a, b: (a - b)
     },
       {NAME: 'ratio', FN: lambda a, b: (a / b)
     },
       {NAME: 'ratio_poly2', FN: lambda a, b: 1.0 / ((5 *
(a ** 2) * (b ** 2)) + (4 * a * b) + 2)
     },
       {NAME: 'coef_ratio', FN: lambda a, b: (a / b)
     },
       {NAME: 'poly2', FN: lambda a, b: (5 * (a ** 2) * (b
** 2)) + (4 * a * b) + 2
     },
       {NAME: 'ratio_poly', FN: lambda x: 1 / (5 * x + 8 *
x ** 2)
     },
       {NAME: 'poly', FN: lambda x: 1 + 5 * x + 8 * x ** 2
     },
       {NAME: 'sqrt', FN: lambda x: np.sqrt(x)
```

```
      },
        {NAME: 'log', FN: lambda x: np.log(x)
      },
        {NAME: 'pow', FN: lambda x: x ** 2
      }
    ]


    def generate_dataset(rows):
        df = pd.DataFrame(index=range(1, rows + 1))

        predictor_columns = []
        y_columns = []

        for f in GENERATED_FEATURES:
            arg_count =
len(inspect.signature(f[FN]).parameters)
            for arg_idx in range(arg_count):
                col_name = "[*]-x[*]".format(f[NAME],
arg_idx)
                predictor_columns.append(col_name)
                df[col_name] = (2 * np.random.random(rows))
- 1
            y_columns.append("[*]-y0".format(f[NAME]))

        idx = 0
        generated_columns = []

        for f in GENERATED_FEATURES:
            col_name = "[*]-y0".format(f[NAME])
            generated_columns.append(col_name)
            arg_count =
len(inspect.signature(f[FN]).parameters)
            a = [df.iloc[:, idx + x] for x in
range(arg_count)]
            df[col_name] = f[FN](*a)
            idx+=arg_count
        return df, predictor_columns, generated_columns


    def generate_report(df, generated_columns):
        report =
pd.DataFrame(index=range(len(generated_columns)))
        y = df.loc[:, generated_columns]
        report['name'] = generated_columns
        report['max'] = np.amax(y, axis=0).tolist()
        report['min'] = np.amin(y, axis=0).tolist()
```

```
       report['range'] = report['max'] - report['min']
       report['mean'] = np.mean(y, axis=0).tolist()
       report['std'] = np.std(y, axis=0).tolist()
       return report


   def generate_predictor_report(df, predictor_columns):
       report =
pd.DataFrame(index=range(len(predictor_columns)))
       y = df.loc[:, predictor_columns]
       report['name'] = predictor_columns
       report['max'] = np.amax(y, axis=0).tolist()
       report['min'] = np.amin(y, axis=0).tolist()
       return report


   def remove_outliers(df, target_columns, sdev):
       for col in target_columns:
           df = df[np.abs(df[col] - df[col].mean()) <=
(sdev * df[col].std())]
       return df


   def main():
       np.random.seed(SEED)

       df, predictor_columns, generated_columns =
generate_dataset(int(NUM_ROWS * 4))

       len1 = len(df)
       df = remove_outliers(df, generated_columns,
OUTLIER_THRESH)
       len2 = len(df)

       print("Removed [*]([*]%) outliers.".format(len1 -
len2, 100.0 * ((len1 - len2) / len1)))

       df = df.head(NUM_ROWS)
       df = df.reset_index(drop=True)
       df.index = np.arange(1, len(df) + 1)
       df.to_csv("/Users/jeff/temp/feature_eng.csv",
index_label='id')

       report = generate_report(df, generated_columns)

       print(report)
```

```
main()
```

Appendix F

Engineered Features for the Data Sets

The AFE algorithm could generate engineered features for the following data sets that increased the neural network accuracy by a statistically significant amount when trained by Encog:

- crx
- hepatitis
- soybean_large
- wine

This appendix provides the features that were engineered by the algorithm. These features are in the form that the algorithm provided them. Note that the percentage operator (%) signifies a genetic programming style protected division operator (Koza, 1992). Protected division returns the value of 1 when the denominator is 0.

No algebraic simplification was performed on the expressions presented in this appendix. The expressions are in exactly the form returned by the algorithm. The top 5 expressions are given, not all were necessarily important to improving the accuracy of the neural network.

Credit Approval Data Set (Crx)

The first engineered feature was:

```
((6+(-(a7-2)+-12.11574089980383))--(-
((a9%7.370904240137026))))
```

The second engineered feature was:

```
((-7.326922223231946%a14)+-17)
```

The third engineered feature was:

```
(a8+11.663433332763784)
```

The fourth engineered feature was:

```
(7.169314926840066%(-8-(-(a7-3)^1)))
```

The fifth engineered feature was:

```
(((3.425788247640814-((3.702263697584752+a4-3)+4))+((a6-0%-
3)+(a6-14*-7)))^(((a6-9--7.496444956829384)+(a1-0*-2))*a6-
5))
```

Hepatitis Data Set (Hepatitis)

The first engineered feature was:

```
(protime%(((-
1^8.509188665805073)+(((antivirals%8.057340918628082)-
1)+((-9.142526195604635*varices-2)+(ascites-0%-
6.569018395243251))))%((histology%spiders-1)-1)))
```

The second engineered feature was:

```
-((-2+(-(spiders-1)%35.192491116187306)))
```

The third engineered feature was:

```
(-((1%(-3.7605168920114496%fatigue-1)))^(((-
4*bilirubin)+fatigue-1)+9))
```

The fourth engineered feature was:

```
-((-2+(-(spiders-1)%38.427838412028905)))
```

The fifth engineered feature was:

```
-((-2+(-(spiders-
1)%(4*(35.70732174870545+((3.5698964173816687*varices-
0)%(varices-2-5)))))))
```

Soybean (Large) Data Set (Soybean)

The first engineered feature was:

```
(-((0.5921349881849763-(fruiting-bodies-
2%0)))%(((7.0813545306629635+((seed-tmt-3*seed-tmt-
1)^7.016568220732647))*(int-discolor-2^((area-damaged-
4^1.9356737945298135)%-18)))*(((((((-
9.789431828678385^4.132015432022388)*(external-decay-
1*(72.99613855528524*((-
8.041427390682856*(2.947873622783181-mold-growth-
0))+2))))%(0.49332060852966997+((6.316274327060544%-(seed-
size-0))^6)))^((38.453229753978654*(-8.396758075416077-
(0+canker-lesion-3)))%((seed-0*(4.187375275907437%(-
3.737282999919236+stem-cankers-
4)))^(((8.19380113073853 2%crop-hist-0)%-
6.024401895787 4435)%12.021301866008182))))^(9.0555303322901
57^temp-0))+(-2*(9%fruit-pods-2)))*((2.152281195372362--
(stem-cankers-0))*(precip-1*2.8026979383297794)))))
```

The second engineered feature was:

```
(-((-0.9189201772638895-(fruiting-bodies-
2%0)))%(((6.9011685373870355+((seed-tmt-3*seed-tmt-
1)^6.484642442989197))*(int-discolor-2^((area-damaged-
4^2.434787503401039)%-18)))*(((((((-
8.926238730978975^3.765680210308123)*(external-decay-1*-
5.955737205802789))%(1.3743879365608411+((6.727787749950141
%-(seed-size-0))^6)))^((38.453229753978654*(-
7.874590346278849-(0+canker-lesion-3)))%((seed-
0*(4.3747690156462955%(((1.443741829421029-(0+(int-
discolor-0*-5.225612923226253)))^(((hail-1^-
5.769380370379435 5)^1.0424275795768527)^(0.1668314181187967
6%(plant-stand-0+0))))+stem-cankers-
4)))^(((8.19380113073853 2%crop-hist-0)%-
6.0244018957874435)%11.070479446663265))))^(8.4216248421574
88^temp-0))+(-2*(9%fruit-pods-2)))*((2.0562327191587917--
(stem-cankers-0))*(precip-1*1.7751408695346078)))))
```

The third engineered feature was:

```
(-((-0.5437368350827702-(fruiting-bodies-
2%0)))%(((7.834703649991203+((seed-tmt-3*seed-tmt-
1)^7.016568220732647))*(9.244435628694996+((-(lodging-1)-
(8*sclerotia-2))^(8%(0.763732880656049-canker-lesion-
0)))))*(((((((-
8.926238730978975^5.062896207808844)*(external-decay-1*-
4.522327547659902))%(1.1146549878212895+((5.673326760938446
%-(seed-size-0))^6)))^((38.453229753978654*(-
8.074311457309028-(0+canker-lesion-3)))%((seed-
0*(4.2620186033338845%(-3.737282999919236+stem-cankers-
4)))^(((7.88938474277853%crop-hist-0)%-
4.488095747846364)%14.202405955494319))))^(7.01438345330483
95^temp-0))+(-2*(9%fruit-pods-2)))*((1.5473942401936414--
(stem-cankers-0))*(precip-1*2.8640081369243258)))))
```

The fourth engineered feature was:

```
(-((0.37806897734595035-(fruiting-bodies-
2%0)))%(((8.226472078599103+((seed-tmt-3*seed-tmt-
1)^5.377881336046352))*(11.818276880555793+((-(lodging-1)-
(8*sclerotia-2))^(8%(2.4707743114800387-canker-lesion-
0)))))*(((((((-
8.972690306696542^3.765680210308123)*(external-decay-1*-
4.24513736228948))%(1.1146549878212895+((6.807335268032035%
-(seed-size-0))^6)))^((38.453229753978654*(-
7.305972062601912-(0+canker-lesion-3)))%((seed-
0*(1.9717851404686058%(-2.723951379378996+stem-cankers-
4)))^(((6.957834989845951%crop-hist-0)%-
6.0244018957874435)%10.78692383813486))))^(8.42162484215748
8^temp-0))+(-2*(9%fruit-pods-2)))*((1.2161900162821047--
(stem-cankers-0))*(precip-1*1.7751408695346078)))))
```

The fifth engineered feature was:

```
(-((-0.5437368350827702-(fruiting-bodies-
2%0)))%(((8.226472078599103+((seed-tmt-3*seed-tmt-
1)^7.016568220732647))*(9.878964913870824+((-(lodging-1)-
(8*sclerotia-2))^(8%(2.4707743114800387-canker-lesion-
0)))))*(((((((-
8.926238730978975^3.765680210308123)*(external-decay-1*-
4.24513736228948))%(1.1146549878212895+((6.316274327060544%
-(seed-size-0))^6)))^((38.453229753978654*(-
7.305972062601912-(0+canker-lesion-3)))%((seed-
0*(4.187375275907437%(-3.737282999919236+stem-cankers-
4)))^(((8.193801130738532%crop-hist-0)%-
6.0244018957874435)%11.070479446663265))))^(8.4216248421574
88^temp-0))+(-2*(9%fruit-pods-2)))*((1.5473942401936414--
(stem-cankers-0))*(precip-1*1.7751408695346078)))))
```

Wine Data Set (Wine)

The first engineered feature was:

```
((((((color_intensity^total_phenols)+0)+72.99971530425738)%
(((1.6767113954095336%alcalinity_ash)^(1-
((flavanoids^(2.7122345849207767--
(malic_acid))^((4%(od28_od315%4.119741319687906))+((-
5%alcohol)*1.2800619275259493)))))*-11.940837228486675))^-
((-(total_phenols)+8)))%(((alcohol%6)*-
0.37222639609811914)+-((flavanoids-ash))))
```

The second engineered feature was:

```
(((magnesium+((7.542468531094034^(total_phenols^2.969043114
577805))-((6--(ash))-(((((5-flavanoids)+(-
1.887330658434184*color_intensity))^-0.018866254325722687)-
-6)+(alcalinity_ash%9))))))^(ash%hue))%(((alcohol%6)*-
0.37222639609811914)+((8.576795897083196^((hue*-
3.268729090560387)^0))%(0%hue))))
```

The third engineered feature was:

```
(((magnesium+((7.542468531094034^(total_phenols^2.969043114
577805))-((6--(ash))-
(alcalinity_ash+(alcalinity_ash%9))))))^(ash%hue))%(((((5.60
1278729430382*(proanthocyanins%-7.381645916985297))-
(magnesium-proanthocyanins))%6)*-
0.37222639609811914)+((8.576795897083196^((hue*-
3.268729090560387)^0))%(0%hue))))
```

The fourth engineered feature was:

```
((-6.191083816011499^-214.93571903026356)+(-27^(-((7-
color_intensity))%((0.10866723540463635-(-
0.2908594216097491%(total_phenols-proanthocyanins)))^-
(((0^total_phenols)%0))))))
```

The fifth engineered feature was:

```
((-15.162959286967478^-8.914653447205758)+((-
18.28953365078993^flavanoids)%((1.1151052593044681+(11083.5
50907581806%(proline+-3)))-2.6617334438560087)))
```

Appendix G

Detailed Experiment Results

The raw data obtained from the experiments in this dissertation are available on the author's Github repository. For more information on obtaining these files, refer to Appendix H, "Dissertation Source Code Availability." The result files are stored as CSV format files:

- report-exp1.csv
- report-exp2.csv
- report-exp3.csv
- report-exp4.csv
- report-exp5.csv
- report-exp6.csv
- workload.csv

The first six files, named report 1-6 are all summarizations of the training cycles that were run. The final workload.csv file, which is much longer, contains the individual cycle results for all experiments that were summarized to produce the individual experiment reports.

The format of each of these files is given in the following sections.

Format for report-exp1.csv

This file holds the results for experiment 1. The min/max/mean errors were reported in RMSE for all rows in this experiment. The fields contained in this file are as follows:

- "experiment" – The experiment run, this is “experiment 1” for all rows in this report.
- "algorithm" – The algorithm being used, either neural network or GP. Also, specifies if this is classification or regression.
- "dataset" – The data set being used.
- "min" – The minimum error.
- "max" – The maximum error.
- "mean" – The mean error.
- "sd" – The standard deviation of this error.
- "elapsed" – The average time to run one cycle of this dataset/algorithm.

Format for report-exp2.csv

This file holds the results for experiment 2. The min/max/mean errors were reported in normalized RMSE for all rows in this experiment. The fields contained in this file are as follows:

- "experiment" – The experiment run, this is “experiment 2” for all rows in this report.
- "algorithm" – The algorithm being used, either neural network or GP. Also, specifies if this is classification or regression.
- "dataset" – The data set being used.

- "min" – The minimum error.
- "max" – The maximum error.
- "mean" – The mean error.
- "sd" – The standard deviation of this error.
- "elapsed" – The average time to run one cycle of this dataset/algorithm.

Format for report-exp3.csv

This file holds the results for “experiment 3”. The min/max/mean errors were reported in normalized RMSE for all rows in this experiment. The fields contained in this file are as follows:

- "experiment" – The experiment run, this is experiment 3 for all rows in this report.
- "algorithm" – The is “ensemble” for either classification or regression, for this experiment.
- "dataset" – The data set being used.
- "min" – The minimum error.
- "max" – The maximum error.
- "mean" – The mean error.
- "sd" – The standard deviation of this error.
- "elapsed" – The average time to run one cycle of this dataset/algorithm.

Format for report-exp4.csv

This file holds the results for experiment 4. The fields contained in this file are as follows:

- "experiment" – The experiment run, this is experiment 4 for all rows in this report.
- "algorithm" – The is “patterns” for this experiment.
- "dataset" – The data set being used.
- "result" – The patterns found.
- "elapsed" – The time it took to run this dataset/algorithm.

Format for report-exp5.csv

This file holds the results for experiment 5. The fields contained in this file are as follows:

- "experiment" – The experiment run, this is “experiment 5” for all rows in this report.
- "algorithm" – The is “importance” for this experiment.
- "dataset" – The data set being used.
- "IMP-W" – How many cycles was the weight-based ranking algorithm stable for.
- "IMP-P" – How many cycles was the perturbation-base ranking algorithm stable for.
- "elapsed" – The time it took to run this dataset/algorithm.

Format for report-exp6.csv

This file holds the results for experiment 6. The min/max/mean errors were reported in normalized RMSE for all rows in this experiment. The fields contained in this file are as follows:

- "experiment" – The experiment run, this is “experiment 6” for all rows in this report.
- "algorithm" – The is “autofeature” for either classification or regression, for this experiment.
- "dataset" – The data set being used.
- "min" – The minimum error.
- "max" – The maximum error.
- "mean" – The mean error.
- "sd" – The standard deviation of this error.
- "elapsed" – The average time to run one cycle of this dataset/algorithm.

Format for workload.csv

The workload.csv file contains the detailed results from all cycles. Most experiment parts are repeated 100 times, and the average error is reported. Because of this, the workload file will have nearly 2,000 lines. The columns reported in this file are as follows:

- "name" – The name of the experiment that was run.
- "status" – The status of this experiment cycle, this should be “complete” for file that has run to completion.

- "dataset" – The data set that was used for this experiment, cycle and algorithm.
- "algorithm" – The algorithm that was used.
- "predictors" – The predictors to use, or empty to use them all.
- "cycle" – The cycle number.
- "result" – The result of this experiment, given in the format specified by that experiment.
- "result-raw" – The raw (RMSE) result from this experiment. This is usually the error.
- "i1" – An experiment specific integer that can be reported.
- "i2" – Another experiment specific integer that can be reported.
- "iterations" – The number of training iterations that were completed for this experiment.
- "elapsed" – The elapsed time (seconds) for this experiment.
- "info" – Any additional information for this experiment.

Appendix H

Dissertation Source Code Availability

The complete source code for this dissertation is available from the author's GitHub repository.[29] The files in this dissertation consist primarily of Java source code, Python source code, and other generated files.

Java Source Code

Java was used to create the actual dissertation algorithm. Java allowed the creation of efficient multi-threaded code that could leverage advanced Amazon AWS instances, such as the c3.8xlarge instance type with 32 processing cores and 60 gigabytes of RAM. The dissertation algorithm could keep all 32 cores at 90%-100% utilization for many data sets. All Java code was located at the following path:

```
    ./gp_java
```

All code can be compiled using the Gradle build system. Running the tasks directive will show the tasks that can be performed.

```
Jeffreys-MacBook-Pro:gp_java jeff$ gradle tasks

...
runAutoFeature – Run the auto feature engineering algorithm
runExperiment1 - Run experiment 1
runExperiment2 - Run experiment 2
runExperiment3 - Run experiment 3
runExperiment4 - Run experiment 4
runExperiment5 - Run experiment 5
runExperiment6 - Run experiment 6
runExperimentAll – Run all experiments.
runGP - Run an experiment
runNeuralXOR - Run an experiment
runSimpleGP - Run an experiment
```

[29] Available from: https://github.com/jeffheaton/phd-dissertation

To run all experiments, use the following command:

```
   gradle runExperimentAll
```

Running all experiments could take hours, depending on the speed of your computer. Running the examples requires a configuration file. This file should be in the user's home directory and named as follows:

```
~/.dissertation_jheaton
```

The file has two configuration lines:

```
   host: macbook-pro
   project: /Users/jeff/temp/dissertation
```

The host line simply specifies the name of the computer that the experiments were run on. The host is written to several output files. The project line specifies where the dissertation report files should be written to.

To run this dissertation's AFE algorithm for any other data sets, run the following command in a directory that has a proper autofeature.json file. Refer to Chapter 3 for the format of this file.

```
   Gradle runAutoFeature
```

Python Source Code

Python source code was used for various tasks. All the final tables and charts, from Chapter 4, were produced using a Python Jupyter Notebook. Another Python file generated the synthetic data set used in Experiment 1. The following Python files were used:

- dissertation_tables.ipynb – Generate the dissertation tables for Chapter 4.
- engineered_feature_dataset.py – Engineered data set for Experiment 1.

Other Files

The complete results from the final run of this dissertation are stored in the file final-output.zip, which is stored in the root of the GitHub repository. The data sets used to evaluate are at the following location:

```
    ~/gp_java/src/main/resources
```

These data sets are bundled into the JAR file that is created to run the experiments.

If you have any questions about running the dissertation code, please contact jeffheaton@ieee.org.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., . . . Devin, M. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*.

Albrecht, R., Buchberger, B., Collins, G. E., & Loos, R. (2012). *Computer algebra: symbolic and algebraic computation* (Vol. 4): Springer Science & Business Media.

Anscombe, F. J., & Tukey, J. W. (1963). The examination and analysis of residuals. *Technometrics, 5*(2), 141-160.

Arnold, K., Gosling, J., & Holmes, D. (1996). *The Java programming language* (Vol. 2): Addison-wesley Reading.

Bahnsen, A. C., Aouada, D., Stojanovic, A., & Ottersten, B. (2016). Feature Engineering Strategies for Credit Card Fraud Detection. *Expert Systems with Applications*.

Balkin, S. D., & Ord, J. K. (2000). Automatic neural network modeling for univariate time series. *International Journal of Forecasting, 16*(4), 509-515.

Banzhaf, W. (1993). *Genetic programming for pedestrians*. Paper presented at the Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93, University of Illinois at Urbana-Champaign.

Banzhaf, W., Francone, F. D., Keller, R. E., & Nordin, P. (1998). *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*: Morgan Kaufmann Publishers Inc.

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., . . . Bengio, Y. (2012). Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*.

Bellman, R. (1957). *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press.

Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and trends in Machine Learning, 2*(1), 1-127.

Bengio, Y. (2013). Representation learning: a review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 35*(8), 1798-1828.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., . . . Bengio, Y. (2010). *Theano: a CPU and GPU math expression compiler.* Paper presented at the Proceedings of the Python for Scientific Computing Conference (SciPy).

Bertsekas, D. P. (1999). Nonlinear programming.

Bíró, I., Szabó, J., & Benczúr, A. A. (2008). *Latent Dirichlet allocation in web spam filtering.* Paper presented at the Proceedings of the 4th international workshop on Adversarial information retrieval on the web.

Bishop, C. M. (1995). *Neural networks for pattern recognition*: Oxford University Press.

Bizer, C., Heath, T., & Berners-Lee, T. (2009). Linked data-the story so far. *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, 205-227.

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent Dirichlet allocation. *The Journal of Machine Learning Research, 3*, 993-1022.

Boisvert, R., Hicklin, J., Miller, B., Moler, C., Pozo, R., Remington, K., & Webb, P. (1998). JAMA: A Java matrix package. *URL:* http://math.nist.gov/javanumerics/jama.

Bottou, L. (2012). Stochastic gradient descent tricks *Neural Networks: Tricks of the Trade* (pp. 421-436): Springer.

Box, G. E. P., & Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological), 26*(2), pp. 211-252.

Breiman, L. (2001). Random forests. *Machine learning, 45*(1), 5-32.

Breiman, L., & Friedman, J. H. (1985). Estimating optimal transformations for multiple regression and correlation. *Journal of the American Statistical Association, 80*(391), 580-598.

Brosse, S., Lek, S., & Dauba, F. (1999). Predicting fish distribution in a mesotrophic lake by hydroacoustic survey and artificial neural networks. *Limnology and Oceanography, 44*(5), 1293-1303.

Brown, B. F. (1998). *Life and health insurance underwriting*: Life Office Management Association.

Brown, M., & Lowe, D. G. (2003). *Recognising panoramas.* Paper presented at the International Conference on Computer Vision (ICCV).

Chea, R., Grenouillet, G., & Lek, S. (2016). Evidence of Water Quality Degradation in Lower Mekong Basin Revealed by Self-Organizing Map. *PloS one, 11*(1).

Cheng, B., & Titterington, D. M. (1994). Neural networks: a review from a statistical perspective. *Statistical science*, 2-30.

Cheng, W., Kasneci, G., Graepel, T., Stern, D., & Herbrich, R. (2011). *Automated feature generation from structured knowledge.* Paper presented at the Proceedings of the 20th ACM International Conference on Information and Knowledge Management.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2015). Gated feedback recurrent neural networks. *arXiv preprint arXiv:1502.02367*.

Coates, A., Lee, H., & Ng, A. Y. (2011). *An analysis of single-layer networks in unsupervised feature learning*. Paper presented at the Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics.

Coates, A., & Ng, A. Y. (2012). Learning feature representations with k-means *Neural Networks: Tricks of the Trade* (pp. 561-580): Springer.

Crepeau, R. L. (1995). *Genetic evolution of machine language software*. Paper presented at the Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, Tahoe City, California, USA.

Cuayáhuitl, H. (2016). SimpleDS: A Simple Deep Reinforcement Learning Dialogue System. *arXiv preprint arXiv:1601.04574*.

Davis, J. J., & Foo, E. (2016). Automated feature engineering for HTTP tunnel detection. *Computers & Security, 59*, 166-185.

Dawkins, R. (1976). *The selfish gene*: Oxford university press.

De Boer, P.-T., Kroese, D. P., Mannor, S., & Rubinstein, R. Y. (2005). A tutorial on the cross-entropy method. *Annals of operations research, 134*(1), 19-67.

Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms*: John Wiley & Sons, Inc.

Dietterich, T. G. (2000). Ensemble methods in machine learning *Multiple classifier systems* (pp. 1-15): Springer.

Diplock, G. (1998). Building new spatial interaction models by using genetic programming and a supercomputer. *Environment and Planning, 30*(10), 1893-1904.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research, 12*(Jul), 2121-2159.

Elman, J. L. (1990). Finding structure in time. *Cognitive science, 14*(2), 179-211.

Filice, S., Castellucci, G., Croce, D., & Basili, R. (2015). Kelp: a kernel-based learning platform for natural language processing. *ACL-IJCNLP 2015*, 19.

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of eugenics, 7*(2), 179-188.

Freeman, M. F., & Tukey, J. W. (1950). Transformations related to the angular and the square root. *The Annals of Mathematical Statistics*, 607-611.

Fukushima, K. (1980). Neocognitron: a self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics, 36*(4), 193-202.

Gao, L., & Hu, Y. (2006). Multi-target matching based on Niching genetic algorithm. *International Journal of Computer Science Network Security, 6*, 215-220.

Garson, D. G. (1991). Interpreting neural network connection weights.

Glorot, X., & Bengio, Y. (2010). *Understanding the difficulty of training deep feedforward neural networks.* Paper presented at the International Conference on Artificial Intelligence and Atatistics.

Glorot, X., Bordes, A., & Bengio, Y. (2011). *Deep sparse rectifier neural networks.* Paper presented at the International Conference on Artificial Intelligence and Statistics.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. Cambridge, MA: MIT Press.

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Gruau, F. (1996). *On using syntactic constraints with genetic programming.* Paper presented at the Advances in Genetic Programming.

Guo, H., Jack, L. B., & Nandi, A. K. (2005). Feature generation using genetic programming with application to fault classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 35*(1), 89-99.

Guyon, I., Gunn, S., Nikravesh, M., & Zadeh, L. A. (2008). *Feature extraction: foundations and applications* (Vol. 207): Springer.

Gybenko, G. (1989). Approximation by superposition of sigmoidal functions. *Mathematics of Control, Signals and Systems, 2*(4), 303-314.

Heaton, J. (2015). Encog: library of interchangeable machine learning models for java and c#. *Journal of Machine Learning Research, 16*, 1243-1247.

Heaton, J. (2016). *An empirical analysis of feature engineering for predictive modeling*. Paper presented at the IEEE Southeastcon 2016, Norfolk, VA.

Hebb, D. O. (1949). The organization of behavior: New York: Wiley.

Hinton, G., Osindero, S., & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computing, 18*(7), 1527-1554. doi:10.1162/neco.2006.18.7.1527

Hinton, G., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science, 313*(5786), 504-507.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. *Diploma, Technische Universität München*.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation, 9*(8), 1735-1780.

Holland, J. H. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks, 4*(2), 251-257.

Ildefons, M. D. A., & Sugiyama, M. (2013). Winning the Kaggle Algorithmic Trading Challenge with the Composition of Many Models and Feature Engineering. *IEICE transactions on information and systems, 96*(3), 742-745.

Janikow, C. Z. (1996). A methodology for processing problem constraints in genetic programming. *Computers & Mathematics with Applications, 32*(8), 97-113.

Jones, E., Oliphant, T., Peterson, P., & al., e. (2001). SciPy: open source scientific tools for Python. Retrieved from http://www.scipy.org/

Jordan, M. I. (1997). Serial order: A parallel distributed processing approach. *Advances in psychology, 121*, 471-495.

Kalchbrenner, N., Danihelka, I., & Graves, A. (2015). Grid long short-term memory. *arXiv preprint arXiv:1507.01526*.

Kanter, J. M., & Veeramachaneni, K. (2015). *Deep feature synthesis: towards automating data science endeavors.* Paper presented at the IEEE International Conference on Data Science and Advanced Analytics (DSAA), 2015. 36678 2015.

Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Knuth, D. E. (1997). *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*: Addison Wesley Longman Publishing Co., Inc.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*: MIT Press.

Kuhn, M., & Johnson, K. (2013). Applied predictive modeling. New York, NY: Springer.

Le, Q. V. (2013). *Building high-level features using large scale unsupervised learning.* Paper presented at the Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on Acoustics.

Lloyd, J. R., Duvenaud, D., Grosse, R., Tenenbaum, J. B., & Ghahramani, Z. (2014). Automatic construction and natural-language description of nonparametric regression models. *arXiv preprint arXiv:1402.4304*.

Lowe, D. G. (1999). *Object recognition from local scale-invariant features.* Paper presented at the The Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999.

Masters, T. (1993). *Practical neural network recipes in C++*: Morgan Kaufmann.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics, 5*(4), 115-133.

McKinney, W. (2012). *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*: O'Reilly Media, Inc.

Miller, J., & Thomson, P. (2000). Cartesian genetic programming *Lecture Notes in Computer Science* (Vol. 1802, pp. 121-132): Springer.

Minsky, M. L., & Papert, S. A. (1969). Perceptrons. an introduction to computational geometry. *Science, 165*(3895).

Mosteller, F., & Tukey, J. W. (1977). Data analysis and regression: a second course in statistics. *Addison-Wesley Series in Behavioral Science: Quantitative Methods*.

Mozer, M. C. (1989). A focused back-propagation algorithm for temporal pattern recognition. *Complex systems, 3*(4), 349-381.

Neshatian, K. (2010). Feature Manipulation with Genetic Programming.

Nesterov, Y. (1983). *A method of solving a convex programming problem with convergence rate O (1/k2).* Paper presented at the Soviet Mathematics Doklady.

Newman, C. L. B. D. J., & Merz, C. J. (1998). UCI repository of machine learning databases.

Ng, A. Y. (2004). *Feature selection, L 1 vs. L 2 regularization, and rotational invariance.* Paper presented at the Proceedings of the twenty-first international conference on Machine learning.

Nguyen, D. H., & Widrow, B. (1990). Neural networks for self-learning control systems. *Control Systems Magazine, IEEE, 10*(3), 18-23.

Nordin, P. (1994). A compiling genetic programming system that directly manipulates the machine code. In K. E. Kinnear, Jr. (Ed.), *Advances in Genetic Programming* (pp. 311-331): MIT Press.

Nordin, P., Banzhaf, W., & Francone, F. D. (1999). Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In L. Spector, W. B. Langdon, U.-M. O'Reilly, & P. J. Angeline (Eds.), *Advances in Genetic Programming* (Vol. 3, pp. 275--299). Cambridge, MA, USA: MIT Press.

Olden, J. D., & Jackson, D. A. (2002). A comparison of statistical approaches for modelling fish species distributions. *Freshwater biology, 47*(10), 1976-1995.

Olden, J. D., Joy, M. K., & Death, R. G. (2004). An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data. *Ecological Modelling, 178*(3), 389-397.

Olshausen, B. A., & Field, D. J. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature, 381*, 607--609.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., . . . Dubourg, V. (2011). Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research, 12*, 2825-2830.

Perkis, T. (1994). *Stack-Based Genetic Programming*. Paper presented at the International Conference on Evolutionary Computation.

Pinkus, A. Z., & Winitzki, S. (2002). YACAS: A do-it-yourself symbolic algebra environment *Artificial Intelligence, Automated Reasoning, and Symbolic Computation* (pp. 332-336): Springer.

Poli, R., Langdon, W. B., & McPhee, N. F. (2008). *A Field Guide to Genetic Programming*: Lulu Enterprises, UK Ltd.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics, 4*(5), 1-17.

Prechelt, L. (1994). Proben1: A set of neural network benchmark problems and benchmarking rules.

Rajaraman, A., & Ullman, J. D. (2011). *Mining of massive datasets*: Cambridge University Press.

Riedmiller, M., & Braun, H. (1992). *RPROP-A fast adaptive learning algorithm.* Paper presented at the Proc. of ISCIS VII), Universitat.

Robinson, A., & Fallside, F. (1987). *The utility driven dynamic error propagation network*: University of Cambridge Department of Engineering.

Rocha, M., & Neves, J. (1999). *Preventing premature convergence to local optima in genetic algorithms via random offspring generation.* Paper presented at the International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems.

Rosenblatt, F. (1962). Principles of neurodynamics.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). *Learning internal representations by error propagation*. Retrieved from

Russell, S., & Norvig, P. (1995). Artificial intelligence: a modern approach.

Scott, S., & Matwin, S. (1999). *Feature engineering for text classification.* Paper presented at the ICML.

Smola, A., & Vapnik, V. (1997). Support vector regression machines. *Advances in neural information processing systems, 9*, 155-161.

Sobkowicz, A. (2016). Automatic Sentiment Analysis in Polish Language *Machine Intelligence and Big Data in Industry* (pp. 3-10): Springer.

Spector, L., & Klein, J. (2006). Trivial geography in genetic programming *Genetic programming theory and practice III* (pp. 109-123): Springer.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research, 15*(1), 1929-1958.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation, 10*(2), 99-127.

Stigler, S. M. (1986). *The history of statistics: the measurement of uncertainty before 1900*: Belknap Press of Harvard University Press.

Sussman, G., Abelson, H., & Sussman, J. (1983). Structure and interpretation of computer programs: MIT Press, Cambridge, Mass.

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). *On the importance of initialization and momentum in deep learning.* Paper presented at the Proceedings of the 30th International Conference on Machine Learning (ICML-13).

Teller, A. (1994). *Turing completeness in the language of genetic programming with indexed memory.* Paper presented at the Proceedings of the First IEEE Conference on Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence.

Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning, 4*(2).

Timmerman, M. E. (2003). Principal component analysis (2nd Ed.). I. T. Jolliffe. *Journal of the American Statistical Association, 98*, 1082-1083.

Topchy, A., & Punch, W. F. (2001). *Faster genetic programming based on local gradient search of numeric leaf values.* Paper presented at the Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001).

Tu, Z., & Lu, Y. (2004). A robust stochastic genetic algorithm (StGA) for global numerical optimization. *IEEE Transactions on Evolutionary Computation, 8*(5), 456-470.

Tukey, J. W., Laurner, J., & Siegel, A. (1982). The use of smelting in guiding re-expression *Modern Data Analysis* (pp. 83-102): Academic Press New York.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math, 58*(345-363), 5.

Van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research, 9*(2579-2605), 85.

Van Rossum, G. (1995). Python tutorial, May 1995. *CWI Report CS-R9526*.

Wang, C., & Blei, D. M. (2011). *Collaborative topic modeling for recommending scientific articles.* Paper presented at the Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

Wang, M., Li, L., Yu, C., Yan, A., Zhao, Z., Zhang, G., . . . Gasteiger, J. (2016). Classification of Mixtures of Chinese Herbal Medicines Based on a Self-Organizing Map (SOM). *Molecular Informatics*.

Werbos, P. (1974). Beyond regression: new tools for prediction and analysis in the behavioral sciences.

Werbos, P. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural networks, 1*(4), 339-356.

White, D. R., Mcdermott, J., Castelli, M., Manzoni, L., Goldman, B. W., Kronberger, G., . . . Luke, S. (2013). Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines, 14*(1), 3-29.

Worm, T., & Chiu, K. (2013). *Prioritized grammar enumeration: symbolic regression by dynamic programming*. Paper presented at the Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, Amsterdam, The Netherlands.

Xue, B., Zhang, M., Browne, W. N., & Yao, X. (2016). A survey on evolutionary computation approaches to feature selection. *IEEE Transactions on Evolutionary Computation, 20*(4), 606-626.

Yu, D., Eversole, A., Seltzer, M., Yao, K., Huang, Z., Guenter, B., . . . Wang, H. (2014). *An introduction to computational networks and the computational network toolkit*. Retrieved from

Yu, H.-F., Lo, H.-Y., Hsieh, H.-P., Lou, J.-K., McKenzie, T. G., Chou, J.-W., . . . Chen-Wei, H. (2011). *Feature engineering and classifier ensemble for KDD cup 2010.* Paper presented at the JMLR: Workshop and Confrence Proceedings 1: 1-16.

Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zhang, W., Huan, R., & Jiang, Q. (2016). Application of Feature Engineering for Phishing Detection. *IEICE transactions on information and systems, 99*(4), 1062-1070.

Ziehe, A., Kawanabe, M., Harmeling, S., & Müller, K.-R. (2001). *Separation of post-nonlinear mixtures using ACE and temporal decorrelation.* Paper presented at the Proceedings of the International Workshop on Independent Component Analysis and Blind Signal Separation (ICA2001).