A STUDY OF ENSEMBLE LEARNING WITH ADABOOST AND NEAT

By

ROBERT SCHUKEI

Bachelor of Science in Computer Science and
Mathematics
Northwest Missouri State University
Maryville, Missouri
2004

Master of Science in Applied Computer Science
Northwest Missouri State University
Maryville, Missouri
2006

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
July, 2017

A STUDY OF ENSEMBLE LEARNING WITH ADABOOST AND NEAT

Dissertation Approved:

Dr. Blayne Mayfield

Dissertation Adviser

Dr. Hokwok Dai

Dr. Johnson Thomas

Dr. Lan Zhu

ACKNOWLEDGEMENTS

The completion of this project would not have been possible without the kind support and help of many individuals and organizations, and would like to extend my sincere thanks to all of them.

I would like to express my most sincere gratitude towards my wife and children for their loving cooperation and encouragement throughout my time working on this project.

I would like to express my special gratitude and thanks to Dr. Blayne Mayfield for giving me much appreciated time and mentoring as I worked towards the completion of this project.

I would like to express my special gratitude and thanks to Dr. Douglas Heisterkamp for giving me much appreciated time, mentoring, and assistance in problem solving as I worked towards the completion of this project.

I would like to express my special gratitude to my committee members, Drs. HK Dai, Johnson Thomas, and Lan Zhu for their time and help on this project.

My thanks and appreciation also go to my colleagues at Baker University and the Oklahoma State University Computer Science Department for their time, knowledge, and use of their equipment in developing the project.

Some of the computing for this project was performed at the OSU High Performance Computing Center at Oklahoma State University supported in part through the National Science Foundation grant OCI-1126330.

Name: Robert Carl Schukei

Date of Degree: July 2017

Title of Study: A STUDY OF ENSEMBLE LEARNING WITH ADABOOST AND NEAT

Major Fields: Computer Science

Abstract: Neural networks, ensemble algorithms, neuroevolution, and genetic algorithms all have shown the ability to solve classification problems in many areas of application. One of the problems in training neural networks is the significant amount of time the training can take. AdaBoost (i.e., Adaptive Boosting) is an algorithm that has been combined with other neural network training algorithms to form ensemble algorithms that have the goal of reducing training times, while retaining the accuracy level of neural network outputs. Another concern with neural networks is that it can be difficult to determine an effective topology for solving particular problems. Neuroevolution can be used to address this issue; neuroevolution uses genetic algorithms to evolve characteristics of a neural network, one of which is its topology. The focus of this study is to investigate whether AdaBoost can be combined with the genetic algorithms of neuroevolution to decrease the time needed to evolve neural networks, and what effects this would have on the accuracy of the results.

TABLE OF CONTENTS

Chapter Page

1 Introduction 1

2 Review Of Literature 3

2.1 Neural Networks . . . 3

2.1.1 Single Neuron . . . 3

2.1.2 Network of Neurons . . . 5

2.2 Neural Network Training . . . 8

2.2.1 Backpropagation . . . 8

2.2.2 Training of the Neural Network . . . 10

2.3 Using the neural network . . . 11

2.4 Ensemble Learning . . . 12

2.4.1 Ensemble Learning . . . 12

2.4.2 Bagging . . . 13

2.4.3 Weighted Majority Voting . . . 13

2.4.4 AdaBoost.M2 . . . 14

2.5 Genetic Algorithms . . . 16

2.6 Neuroevolution . . . 18

Chapter Page

2.6.1 NEAT 19
2.7 Cross-Validation 20

3 Methodology **21**
3.1 Overview 21
3.2 Experiment 23
3.2.1 Decrease of Learning Time 24
3.2.2 Increase of Accuracy 24
3.2.3 Overall Accuracy 25

4 Results **26**
4.1 Results for the Spambase Dataset 26
4.2 Results for the Adult Dataset 31
4.3 Results for the Tic-Tac-Toe Endgame Dataset 33
4.4 Results for the Breast Cancer Wisconsin (Diagnostic) Dataset 36
4.5 Accuracy Runs 37
4.6 Overall Observation 40

5 Conclusion and Future Work **41**

Bibliography **43**

A Adult Dataset Percent Correct Comparison Graphs **47**

B Spambase Dataset Percent Correct Comparison Graphs **60**

C Tic-Tac-Toe EndGame Dataset Percent Correct Comparison Graphs **73**

D Breast Cancer Wisconsin (Diagnostic) Dataset Percent Correct Comparison Graphs **86**

LIST OF TABLES

Table Page

1 5

2 22

LIST OF FIGURES

Figure Page

1 3

2 4

3 6

4 7

25 47

26 48

27 48

28 49

29 49

30 50

31 50

32 51

33 51

34 52

35 52

36 53

37 53

Figure Page

38 54
39 54
40 55
41 55
42 56
43 56
44 57
45 57
46 58
47 58
48 59
49 60
50 61
51 61
52 62
53 62
54 63
55 63
56 64
57 64
58 65
59 65
60 66
61 66
62 67

Figure Page

63 67
64 68
65 68
66 69
67 69
68 70
69 70
70 71
71 71
72 72

73 73
74 74
75 74
76 75
77 75
78 76
79 76
80 77
81 77
82 78
83 78
84 79
85 79
86 80
87 80

Figure Page

88 81
89 81
90 82
91 82
92 83
93 83
94 84
95 84
96 85
97 86
98 87
99 87
100 88
101 88
102 89
103 89
104 90
105 90
106 91
107 91
108 92
109 92
110 93
111 93
112 94

Figure Page

113 94
114 95
115 95
116 96
117 96
118 97
119 97
120 98

Chapter 1

Introduction

There currently are several different techniques that can be used to solve classification problems; a classification problem is one in which a collection of items must be partitioned into discrete classes based on some criteria. Some of the techniques that can be used to solve classification problems include neural networks, neuroevolution, and ensemble algorithms.[1, 2] Artificial neural networks in conjunction with a learning algorithm can be used to simulate any continuous function.[3] If an artificial neural network has at least one hidden layer, then it can simulate any discontinuous function, as well.[3] Though artificial neural networks are a very powerful tool, the main difficulties with them are the choice of a correct network topology and the length of time needed to train the artificial neural network.

One possible solution to decrease the training time is to use artificial neural networks as part of an ensemble algorithm. An ensemble algorithm is a meta-algorithm that joins learning algorithms in some way that combines the solutions of the component learning algorithms. The ensemble learning algorithm that is chosen can affect the training time and accuracy of the component algorithms. For example, boosting ensemble algorithms (i.e., those ensemble algorithms that incorporate a boosting technique) have an effect on accuracy and training times.

Traditionally, choosing a network topology has been a combination of experience and trial and error; more recently, neuroevolution has been used to automate topology selection. Neuroevolution was created from the idea of evolving different attributes of artificial neural networks, including the network topology. This approach exchanges the time and tedium

needed for a human to design a good topology for the computational time needed to evolve the topology.

One thing that has not been taken into consideration is whether it is possible to join neuroevolution together with an ensemble algorithm. This could allow the benefit of both evolving the neural network and the joining of several algorithms together to form a final decision that may be a more accurate and may involve less training time.

NeuroEvolution of Augmenting Teopologies (NEAT) is one example of a neuroevolution algorithm.[4, 5, 6, 7] Adaptive Boosting (AdaBoost)[8, 1] is one example of an ensemble algorithm; that is, it requires another algorithm to do its job. In this research experiments will be performed to compare the effectiveness of NEAT to that of AdaBoost with NEAT.

The data that will be used in these experiments is drawn from the UCI machine learning repository.[9] The four datasets chosen are: Adult dataset (information about salaries), Spambase dataset (information about spam and ham emails), Breast Cancer Wisconsin (Diagnostic) dataset, and Tic-Tac-Toe Endgame dataset.[9] Since these datasets are small, ten-fold cross-validation techniques will be applied to the data; this will permit all data points to be used for training, validation and testing of the artificial neural networks at some time, thus improving the generalization of the classification.

Experiments will be run on the Cowboy cluster of the Oklahoma State University High Performance Computing Center. Results will be collected and analyzed using statistical techniques to try to determine whether using AdaBoost with NEAT provides benefits over using NEAT alone.

Chapter 2

Review Of Literature

2.1 Neural Networks

2.1.1 Single Neuron

Artificial Neural Networks or, more simply, Neural Networks are a programmer's attempt to mimic the properties of biological neurons.[10, 11] Both biological and artificial neural networks are made up of multiple individual neurons. Figure 1 provides an example of a computer representation of a neuron. A basic computer-modeled neuron is made up of inputs, a possible bias value, weights on the both the inputs and the bias, a summation function, an activation function, and an output. An example of a computer neuron with a bias can be seen in Figure 2. Each component of the neuron may play an important role in the overall function of the neuron.

The inputs are what determine how the neuron is going to be activated; a neuron must

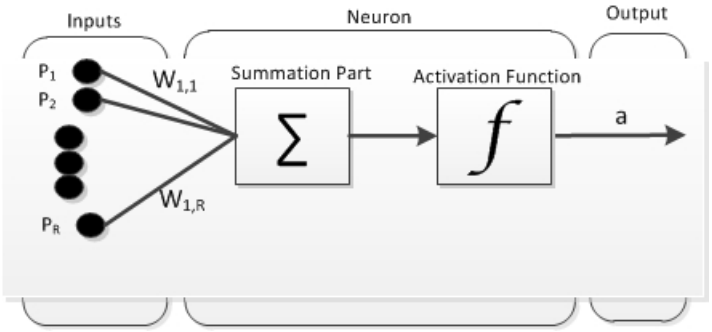

Figure 1: A Single Neuron with out Bias Example

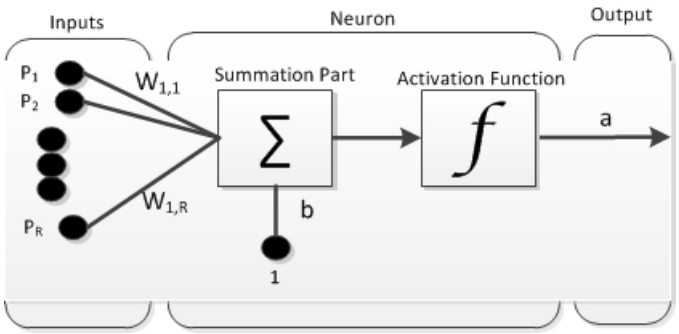

Figure 2: A Neuron with a Bias Example

have at least one input. Each input - as well as the bias - has a weight associated with it; this weight is multiplied by the input/bias values. The bias can be treated just like an input, but always has a value of one that is multiplied by the corresponding weight; in some documentation the weight of the bias is known as the bias.[3, 10] This allows the value of the bias multiplied by the weight to scale the summation value without affecting the other inputs. Whether to include a bias is up to the designer of the neural network, and an example of a neuron without a bias can be seen in Figure 1. The weighted inputs and the weighted bias then are fed into the summation function; the summation function usually just adds together the weighted inputs and bias, but other summation functions could be used in its place. The output of the summation function feeds into the activation function, which will then generate the output of the neuron.

While the inputs, bias, and weights can affect the outcome of the neuron, some say the most important decision in designing a neuron is the choice of its activation function. The activation function determines the output characteristics of the neuron. A list of commonly-used activation functions is found in Table 1. While the list of possibilities may seem long, the correct choice can be relatively simple since the choice is based on the desired output characteristics.

A neuron models a function from its input values to its output value. By adjusting the weights of the neuron, the characteristics of this function can be changed. If one had to adjust these weights manually, it could be very difficult and time consuming to arrive at a

Name of Function	Input	Output
Hard Limit	input < 0 input ≥ 0	output = 0 output = 1
Symmetrical Hard Limit	input < 0 input ≥ 0	output = -1 output = 1
Linear	input	output = input
Saturating Linear	input < 0 $0 \leq$ input ≤ 1 input > 1	output = 0 output = input output = 1
Symmetric Saturating Linear	input $<$ -1 -1 $\leq$ input ≤ 1 input > 1	output = -1 output = input output = 1
Log-Sigmoid	input	output = $\frac{1}{1+e^{-n}}$
Hyperbolic Tangent Sigmoid	input	output = $\frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}$
Positive Linear	input ≤ 0 input > 0	output = 0 output = input

Table 1: Activation Functions

function with desired characteristics. Instead, the weights usually are adjusted automatically through a process by which the neuron learns the function. One possible learning process will be described in a later section.

2.1.2 Network of Neurons

While a single neuron is not all that powerful by itself, it is possible to link neurons together in layers and link layers together. The combination of the neurons is what makes up a neural network. There are different ways to describe neural networks. Some descriptions of neural networks count the network input as a layer, while other descriptions do not consider the input as a layer.[10, 12] The description of neural networks in this paper will be as described in *Neural Network Design* by Hagan, Demuth, and Beale where the inputs are not counted as a layer.[10]

While it is possible to create a single layer neural network the limitations of this type of network does not make them feasible for most applications.[10, 13, 14] Minsky and Papert proved that a single layer neural network can be used only on very narrow and special classes

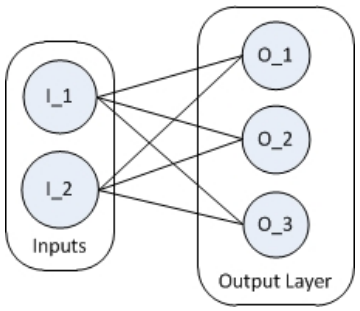

Figure 3: A Single Layer Perceptron Neural Network Example

of problems, those that are linearly separable.[13, 14] The biggest limitation at the time of *Perceptrons* by Minsky and Papert is that no one had figured out how to use different training algorithms on more than a single layer.

It is important to understand the single layer networks and their learning rules to move on to multilayer networks. One type of single layer neural network is known as the Perceptron Network, which uses the perceptron learning rule and is made up of a specific type of neurons named perceptrons. The perceptron was first introduced by Frank Rosenblatt in *Psychological Review*, November 1958 and uses the hard limit activation function with an optional bias.[10, 15] An example of a single-layer perceptron network, with two input values and three output perceptrons can be seen in Figure 3. The perceptron is a single layer neural network to make decisions between linearly separable classifications. The possibility of adding hidden layers between the input neurons and the output layer removes the limitations of the neural network that were pointed out by Minsky and Papert for a single-layer network. The first algorithm that allows for all layers to be trained was not realized until the late 1970's to early 1980's.[10] This algorithm is known as the backpropagation algorithm, which makes it possible for any size of neural network to be trained as long as the activation function for each neuron has a derivative. Note that not all of the activation functions in Table 1 have derivatives, including the hard limit activation function which is the activation function for the perceptron neural network.

An example of a two-layer neural network can be seen in Figure 4; this example of a two-layer perceptron network has three input values, four neurons in a single hidden layer,

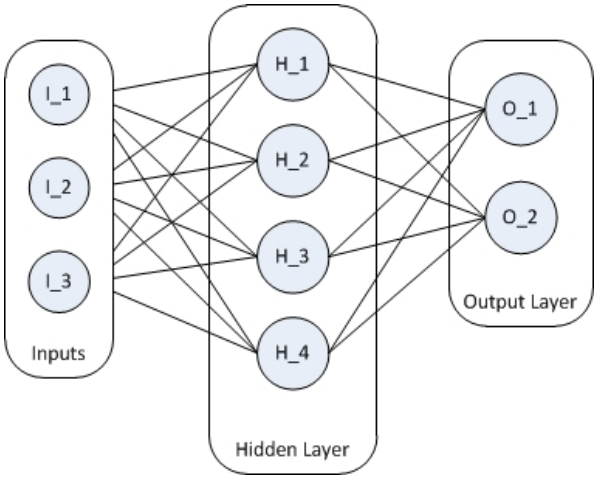

Figure 4: A Two Layer Neural Network Example

and two output neurons in the output layer. Any layer included in the neural network that is not input values or the output layer is considered to be a hidden layer.[10] While it is possible and extremely likely that more hidden layers can be useful to help solve a problem, it has been proven that a single hidden layer can be trained to be equivalent to multiple hidden layers if enough neurons are in the hidden layer, and with enough training data.[13]

While the layout of the neural network is largely made up from experience or trial-and-error, the number of input nodes and output nodes are completely determined by the data being classified and how the program represents the data. The number of inputs is the number of dimensions a single input vector has. The number of outputs is a little harder to figure out since it is possible to have only a single output for any classification, or to have an output neuron for each classification. It is possible to train a neuron to give an output value in a certain range that is used to determine the classification of the input; or have a neuron in the output layer for every classification. It is easier to train the multiple neurons in the output layer. This way, instead of forcing the neural network to learn a value range for a given classification, the neural network learns to have a single output value be high and all other outputs are low for different classifications.

2.2 Neural Network Training

As was seen in the previous section most data is given as pairs that consist of a vector of inputs and a separate vector of corresponding expected outputs, with both the inputs and outputs being continuous or discrete values. Working with both continuous and discrete datasets allows neural networks the ability to work on most datasets. Some of the biggest problems with neural networks are having enough data to train with, determining when to stop training, and deciding how much of the data should be used to train the neural network. Normally when training neural networks, the data is split into either 80%/10%/10% or 70%/15%/15% partitions for training, verification, and testing, respectively. Only the training set actually is used to train the neural network, while the validation set is used to determine when to stop training the neural network. The testing set should never be used in training and is given as input to the neural network only after the network has finished training to test if it can work on data that it has never seen before. This verifies that the neural network is ready to be used on unknown data. It is important that each of the data set partitions are a representative sample of the overall data.

2.2.1 Backpropagation

The actual training method that normally is used on neural networks is the backpropagation algorithm. As mentioned in the previous section one of the main requirements of the method is that all of the activation functions used must have a derivative.[10] For the backpropagation algorithm to work it is necessary to first find a result from one input of the training data set. This is achieved by feeding the input values into the neural network and calculating the result of the first layer by processing the input data in that layer's neurons. After the first layer activation values have been calculated, those are then used as input values to the next layer. This continues until the output layer results have been calculated.

Once the neural network results have been calculated it is possible to then apply the

backpropagation algorithm. The formulas that are shown and explained are for the steepest decent backpropagation. The first step of the steepest decent backpropagation algorithm is to propagate the sensitivity of each neuron backwards through the neural network. This algorithm starts by first finding the error of all output neurons compared to expected value given from the data set. The sensitivity or significance of each output neuron is equal to the equation:

$$s^M = -2 * f'^M * (t - a) \tag{2.1}$$

where s^M is the significance of an output neuron, f'^M is the derivative of the activation function of the specific output neuron, and $(t - a)$ is the calculated error for the output neuron. All neurons that are not output neurons have their significance values calculated as

$$s^M = f'^M * w^{M+1} * s^{M+1}. \tag{2.2}$$

where f'^M is the derivative of the activation function evaluated at that result. w^{M+1} is the weight of the outgoing function to the next specific neuron, and s^{M+1} is the significance of the next neuron associated with the given weight. If a neuron has more than one output connection to neurons in the next layer then its significance is based on the weighted sum of the significances along each output connection. This allows for the error to propagate back to the neurons that uses the input values.

After the significance for each neuron has been calculated, the weights in the neural network can be adjusted to learn a better fit to the given data. The equation for updating the weights not associated with a bias input is given by

$$(w_{new})^M = (w_{old})^M - (\alpha * s^M * a^{M-1}) \tag{2.3}$$

Where s^M is the significance of this neuron, a^{M-1} is the input from the previous layer for this specific weight, and α is a learning constant,known as the learning rates. In some

versions of the backpropagation algorithm α can be changed as the algorithm is ran. For more information about changing the learning rate of the backpropagation algorithm please see *Neural Network Design* by Hagan, Demuth, and Beale.[10] The equation for updating the bias weight is very similar, but ignores the input value from the previous layer since it is always going to be a one. The bias weight equation update is

$$(b_{new})^M = (b_{old}) - \alpha * s^M \tag{2.4}$$

This gives the basic overview of steepest descent backpropagation algorithm, and shows how the output values are first calculated, and then the errors of the output from the expected output are back-propagated through the neurons for the learning to take place. There are several other shortcuts or changes that can be made to the steepest decent backpropagation algorithm; for a listing of these; look at *Neural Network Design* by Hagan, Demur, and Beale.[10]

2.2.2 Training of the Neural Network

The previous section discussed how to train a neural network using the steepest descent backpropagation algorithm using a single data point. This same process can be used for all of the data points in a data set. When all data points are fed forward and back propagated through the network one time, this is known as a training epoch or training generation. A single epoch is normally not enough to train a neural network to learn a function, but will take several epochs for a network to be trained. The order in which the training data is presented can also affect the length of time the neural network takes to train, but the presentation order will not affect the end result of the training.

This raises the question of how to tell when the neural network is done training. It is expedcted that error values should continually decrease as the number of epochs grows. Yet, that just shows that the neural network is learning those specific data points, and

gives no indication of whether the neural network can recognize other data points that are not in the training set, but still be valid data points. This is why the data set should be broken into three different partitions as mentioned in Section 2.2.1. It is possible to perform just the feedforward step on the validation set to retrieve output values, and not perform the backpropagation step. This will provide for the ability to view the error values of not only the training set, but also the validation set. Thus, if the neural network is learning correctly, both the training error and validation error should decrease over time. Eventually, after several epochs of decreasing error in both the training and validation sets, training set error values will continue to decrease, but the validation set error will start to increase in value. This trend is known as overfitting of the training set.[10] This means that the neural network actually has been overtrained and should be reverted back to the last epoch where the validation set was at a minimum error values.

2.3 Using the neural network

After training and validation, neural networks must be tested to see if they work as they should. This is the reason for the third section of the testing data as mentioned in Section 2.2.1. This allows the user to test whether the trained neural network works on data it has not seen. Part of the problem is that the neural network only works as well as the data set with which it was trained on.

One of the biggest problems that can arise is trying to use the neural network on data points that are outside of the training data set area. It is possible to extrapolate a little into areas that were not covered by the data points, but overall there is no guarantee for the extrapolation to be correct.[16] This shows that the user should only use a trained Neural Network on data points in the same area that was covered in the training set.

2.4 Ensemble Learning

2.4.1 Ensemble Learning

Previous sections describe how a neural network learns from known input and output vectors to form a hypothesis to test unknown vectors of input. When discussing neural networks and ensemble learning techniques a hypothesis is a trained neural network. A downfall of neural networks is the length of time it takes to create and test a single hypothesis. With ensemble learning, it is possible to join several weak hypotheses to make a hypothesis that is stronger than any of the individual, weak hypotheses. The formal definition of a weak hypothesis is a function that is designed to learn and is correct a little more than $\frac{1}{|k|}$ of the time, where $|k|$ is number of possible output classifications.[3, 17, 18] A strong hypothesis is one that is correct arbitrarily more than $\frac{1}{|k|}$ of the time.[3, 17, 18] In other words, one weak hypothesis is better than random guessing of the output.

The training of each individual hypothesis/neural network for ensemble learning does not have to be as accurate as a single hypothesis without ensemble learning. With most ensemble learning algorithms, the hypotheses that are joined together are classified as weak hypotheses, which are then joined together to create a strong hypothesis that is very close to producing the correct outputs.[17] When training neural networks our usual goal is a strong hypothesis; but if all that can be achieved is a weak hypothesis, several of these could be joined to form a strong hypothesis. It could be possible that generating a number of weak hypotheses and forming them into a strong hypothesis will take about the same amount of time as generating a strong hypothesis; but combining weak hypotheses could create a better strong hypothesis then the generation of a single, strong hypothesis. It is also possible to use ensemble methods on already-trained hypotheses; that could increase the accuracy of the final hypothesis. There are several different ensemble learning algorithms, including bagging, boosting, weighted majority voting, and rank voting.[1, 3, 19, 8] A brief introduction to the first three of these ensemble methods can be seen in the following sections.

2.4.2 Bagging

As has been stated, there are several different ensemble techniques, and one of the earliest was bagging (bootstrap aggregating).[3, 20] Bagging allows the generation of multiple hypotheses (which the author calls predictor hypotheses) and their aggregation to form a single predicting hypothesis. [20] The multiple hypotheses are formed by using bootstrap replicas (defined below) of the original learning data set, and creating a hypothesis for each of the replica data sets.[20] A bootstrap replica is created from the training data set by sampling with replacement from the original training data set, such that the bootstrap replica is the same size as the original training data set.[20, 21, 22] Bootstrap replicas allow for the creation of multiple training data sets from a single training data set. The bootstrap data sets then can be used to approximate the confidence interval of the original data set, if needed, or to find out which output occurs most often amongst the different data sets for use as the aggregation result.[20, 22] For a more in-depth look into bagging, see *Bagging Predictors* by Leo Breiman.[20] For more information about the use of bootstrap replicas to find confidence intervals, look at *An Introduction to the Bootstrap* by Bradley Efron and Robert Tibshirani.[22]

2.4.3 Weighted Majority Voting

Weighted majority voting is an ensemble learning algorithm based on the weighted majority algorithm used primarily for binary classification problems. This ensemble algorithm starts out with a pool of hypotheses, where each hypothesis has a weight of one associated with it.[3, 23, 24] These hypotheses are given the same input vector and calculates their output vectors. The hypotheses are separated into two partitions, q0 and q1, based on whether their outputs are 0 or 1, respectively. The sum of weights of the hypotheses in each partition is calculated, and the partition with the larger sum dictates the calculated value of the ensemble hypothesis; that is, if q0 has the larger sum, the calculated value of the ensemble hypothesis is 0, and the calculated value of the ensemble hypothesis is 1 otherwise.

When the calculated output of the ensemble hypothesis does not match the expected output associated with the input vector, the weight of each hypothesis from the partition with the larger sum is multiplied by β, which is a value between zero and one.[23] This decreases the weights that would be summed together in future instances for those hypotheses, which penalizes the hypotheses that continually make mistakes. For more detailed information on weight majority voting and choosing a value for β see *The Weight Majority Algorithm* by Littlestone and Warmuth.[23]

2.4.4 AdaBoost.M2

Algorithm 1 AdaBoost

function AdaBoost(examples, L, M)
  input: *examples*, set of N labeled examples $(x_1, y_1), ..., (x_N, y_N)$
    L, a learning algorithm
    M, the number of hypothesis in the ensemble
  local variables: $\mathbf{w}$, a vector of N example weights, initially $1/N$
    $\mathbf{h}$, a vector of M hypotheses
    $\mathbf{z}$, a vector of M hypothesis weights
  for $m = 1 \rightarrow M$ **do**
    $\mathbf{h}[m] \leftarrow L(examples, \mathbf{w})$
    $error \leftarrow 0$
    for $j = 1 \rightarrow N$ **do**
      if $h[m](x_j) \neq y_j$ **then**
        $error \leftarrow error + \mathbf{w}[j]$
      end if
    end for
    for $j = 1 \rightarrow N$ **do**
      if $h[m](x_j) = y_j$ **then**
        $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \bullet error/(1 - error)$
      end if
    end for
    $\mathbf{w} \leftarrow Normalize(\mathbf{w})$
    $\mathbf{z}[m] \leftarrow log(1 - error)/error$
  end for
  return $Weighted - Majority(\mathbf{h}, \mathbf{z})$
end function

Boosting is a form of ensemble learning that produces an accurate hypothesis by com-

bining rough or weak hypotheses together. This process is done by using a labeled training set (i.e., a set of input and expected output pairs), and the more often a weak hypothesis is correct, the more influence that weak hypothesis has on the overall combined hypothesis.[8] An important feature of boosting is that the weak hypotheses do not have to be any better that random guessing. Boosting algorithms learn by evaluating and ranking which hypotheses are correct more often.[8] One boosting algorithm that commonly is used is the Adaptive Boosting Algorithm(AdaBoost). The pseudocode of the AdaBoost algorithm can be seen in Algorithm 1. (Note that the function **Weighted-Majority** forms a hypothesis as described in Section 2.4.3.) One perk of using the Adaboost algorithm over other algorithms is that as M (the number of hypotheses being used) increases, the probability that the ensemble hypothesis is going to be correct also increases.[3]

Algorithm 2 AdaBoost.M2

function AdaBoost.M2(examples, labels)

input: *examples*, set of N labeled examples $(x_1, y_1), ..., (x_N, y_N)$

labels, set of k labels of possible outputs

Init: $\mathbf{B} = \{(i,\ y) : i \in \{1, ..., N\},\ y \neq y_i$

$D_1(i,\ y) = 1/|B|$ for all $(i,\ y) \in B$

Repeat:

1. Train Neural network with respect to distribution D_t and obtain hypothesis h_t
2. calculate the psuedo-loss of h_t:
$$\epsilon_t = \frac{1}{2} \sum_{(i,y) \in B} D_t(i,y)(1 - h_t(x_i, y_i) + h_t(x_i, y))$$
3. $\beta_t = \epsilon_t/(1 - \epsilon_t)$
4. update distribution D_t
$$D_{t+1}(i,\ y) = \frac{D_t(i,y)}{Z_t} \beta_t^{\frac{1}{2}(1 + h_t(x_i, y_i) - h_t(x_i, y))}$$

where Z_t is a normalization constant

Output: final hypothesis
$$f(x) = \arg \overset{max}{y \in Y} \sum_t (log \tfrac{1}{\beta_t}) h_t(x, y)$$

end function

While the AdaBoost algorithm is powerful by itself, improvements have been made to the algorithm depending on which learning algorithm will be used.[1] The AdaBoost.M2 algorithm was designed specifically to be used with neural network classification problems.[1,

8] A copy of the pseudocode for the AdaBoost.M2 algorithm can be seen in Algorithm 2.

2.5 Genetic Algorithms

Genetic algorithms is a technique that develops systems that learn and explores a state space using processes similar to biological organisms.[3, 12, 25] It uses a string representation of the state space to represent individual genes (as any species has) and uses them to perform reproduction to form new individuals. There are three things that genetic algorithms require: (1) There is some fitness function for each individual that helps determine the possible influence of the individual on future generations; (2) There is a mating operation that determines the next generation of individuals; and (3) There are genetic operations that form the genetic code of offspring based on the genetic code of the parent(s).[25] Pseudocode for a general genetic algorithm can be seen in Algorithm 3.

Genetic algorithms start out with a random population of individuals that are represented in some string format, usually in a bit string.[3, 12, 26] This string representation is considered to be an individual that will be used in the reproduction phase of the genetic algorithm. These individuals are processed to determine which individual is the most fit using a fitness function. Those individuals with better fitness have a better chance of reproducing, but this is not guaranteed.[3, 25, 26]

The normal reproduction, considered by some to be the most important part of a genetic algorithm, is called crossover, which allows two individuals to mix in some way to form an individual of the next generation.[26] It is possible that since the individuals are chosen at random, both parents are actually the same individual. The most common form of crossover is known as a single-point crossover, which that takes the first part of one parent and concatenates this with the second part of the other parent.[3, 12, 26] This crossover completes the second and third requirements for genetic algorithms, given above. The point where the crossover occurs in the string representation of the parents normally is chosen at

a random point for each parent pair. A single-point crossover is the most common type of crossover, but there are other types, including the multi-point crossover. The only difference between a single-point crossover and a multi-point crossover is that in a multi-point crossover multiple points are chosen.

Multi-point crossover creates a new individual by concatenating the first part of the first parent (from its beginning to the first point), with the second part of the second parent (from the first point to the second point), with the third part from the first parent (from the second point to the third point), and so on.

While crossover is the main change that occurs during reproduction, there is a chance that random mutation occurs in any of the individuals in the new generation. This mutation changes one or more of the characteristics or genes in the string representation to another valid character.[3, 12, 26]

Algorithm 3 Genetic Algorithm

```
function GENETIC ALGORITHM(population, Fitness-FN)

    input: population, a set of individual
           Fitness-FN, a function that measures the fitness of an individual
    local variables: new_population, an empty set

    repeat
        for i = 1 →Size(population) do
            x ← RandomSelection(population, Fitness-FN)
            y ← RandomSelection(population, Fitness-FN)
            child ← Reproduce(x, y)
            if (small random probability) then
                child ← Mutate(child)
            end if
            add child to new_population
        end for
    until (some individual is fit enough, or enough time has elapsed)
    return the best individual in population according to Fitness-FN
end function
```

2.6 Neuroevolution

While genetic algorithms can be used on many types of problems, researchers struggled to find an efficient way to use this technique to train or create neural networks. Yet researchers finally were able to figure out how to use genetic algorithms with neural networks, and the field of neuroevolution was formed. The original neuroevolution technique used genetic algorithms to change only the weights of a neural network, and not its topology.[7] The NeuroEvolution of Augmenting Topologies (NEAT) was an original idea by Kenneth Stanley describing a way to evolve the topology and weights of neural networks using genetic algorithms.[4, 5, 6, 7] Neuroevolution searches through the state space of all neural networks to find a neural network that succeeds at solving the problem at a user specified level of success.

There are two main types of neuroevolution techniques currently in use today: direct encoding and indirect encoding.[7, 27] Direct encoding represents both the weights and nodes of a neural network in a form of the string representation used for genetic algorithms.[7, 27] Indirect encoding takes a different viewpoint and only states rules about how the neural network can be formed.[7, 27] For several years, neuroevolution had problems evolving the topology of a neural network. A crossover could possibly create an invalid network where there is no path between the input values and output nodes, or the neural network could forget parts of previously-learned information. An example of an evolved neural network forgetting part of what it had previously learned would be the following: if one of its parent neural networks learned the inputs in the format A, B, C and a second parent learned them in the order C, B, A, then a single-point crossover might produce a child with the format A, B, A or C, B, C, meaning the new neural network would have forgotten either A or C.

2.6.1 NEAT

While there are different neuroevolution techniques available for use, the algorithm that will be used for these experiments is a form of direct encoding known as NEAT. Part of the reason for using NEAT in the experiments is that Kenneth Stanley and Risto Miikkulainen found a way around both of the previously-stated limitations of neuroevolution, by finding a way to evolve the structure of a neural network.[7] The NEAT algorithm allows neural networks to evolve new nodes, links (connections between nodes), and weights, all while keeping a historical marking for when new nodes and new links are created. This historical tracking allows the NEAT algorithm to determine where different, evolved neural networks in a population can perform a crossover. For example if two links in different neural networks have the same historical markings, then the two neural networks can perform a crossover reproduction at the link that they have in common to form a child neural network. A broad overview of the NEAT can be seen in Algorithm 4. For more detailed information on the NEAT algorithm, see *Efficient Evolution of Neural Networks Through Complexification* by Dr. Kenneth Stanley.[4]

Algorithm 4 NEAT Algorithm

function NEAT(examples)

input: *examples*, a set of of N labeled examples (x_1, y_1), ... , (x_N, y_N) with labels $y_i \in Y = 1, ...k$

local variables: **P** a random population of neural networks in the state space of the problem.

repeat

Evaluate the **P** neural networks to calculate there fitness

Move the **P** neural networks into different species groups

Perform a generation evolution on the **P** neural networks using crossover and mutation.

until (some individual is fit enough, or enough time has elapsed)

return the best individual in population according to **Fitness-FN**

end function

2.7 Cross-Validation

One way to verify that a hypothesis is correct is known as the holdout method, as described in Section 2.2.1. This technique specifically sets aside a portion of the data that is not used for training the neural network; instead the data set aside is used for testing the accuracy of the hypotheses. A second verification technique normally used on small data sets is called cross-validation. In cross-validation, a data set is split into k mutually exclusive data sets.[28, 29] Then, the different partitions can be recombined to form k different training and testing sets. The most popular values of k are 1, 5, 10, and n; where n is the size of the original data set.[29] After recombining the different partitions into the k different training and testing data set pairs, the training data sets are used for training at least k different hypotheses. Each hypothesis then can be tested(validated) using the testing data set that corresponds to the data set with which the hypothesis was trained. Since there are at least k hypotheses that are trained, cross-validation takes a longer time to train and validate than the holdout method. For more information on both holdout and cross-validation, see *A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection.*[28]

Chapter 3

Methodology

Up until now, the information presented has been about the past of neural networks, neuroevolution and ensemble algorithms. The question now is whether the different algorithms already presented can be joined in new or different manners then has been done previously. An interesting question, in both an academic sense and for real world problems, is whether it is beneficial to use an ensemble algorithm in conjunction with NEAT. In the past, neural networks have been trained to solve classification problems such as identification of breast cancer, classification of correspondence or documents, and speech segregation. [30, 31, 32, 33] This study is an investigation of whether NEAT can work in conjunction with the AdaBoost algorithm to decrease the time needed to evolve neural networks without a loss in accuracy, or to increase the accuracy of neural networks without significantly increasing the evolution time of such networks.

3.1 Overview

To start the experiment, a copy of the C++ code for NEAT was obtained from the Neural Network Research Group at the University of Texas, and C++ code was written for the implementation of the AdaBoost ensemble method and traditional Artificial Neural Networks.[34] This code base is what the different data sets (Adult, Spambase, Breast Cancer Wisconsin - Diagnostic, and the Tic-Tac-Toe Endgame Data Sets) were tested on.[30, 31, 32, 33] Each data set can be obtained from the UCI (University of California, Irvine) machine

Dataset	Percentage Correct Value
Adult Dataset	80%
Spambase Dataset	90%
Breast Cancer Wisconsin (Diagnostic) Dataset	95%
Tic-Tac-Toe Endgame Dataset	75%

Table 2: Dataset Correctness Target Percentage

learning repository.[9]

After obtaining the data sets from UCI, the datasets needed to be processed to make them ready to be used with our experimental code. One example of this was to change the expected output from a category name to a number representing that category. Once the preparatory work on the datasets was complete, the datasets were split up using a ten-fold cross-validation, with each fold being split in a 70%-20%-10% way representing training, validation, and testing datasets, respectively. (For more information on cross-validation see Section 2.7.) After the ten-fold cross-validation datasets were created, the data was normalized to create zero-mean, unit-variance datasets based upon the training and validation datasets.

After the data was preprocessed, there was a need to determine the percentage of correct values to represent the stopping condition for each dataset. These percentages were based upon historical data provided with the datasets and those found in the publications cited here.[9, 35, 36] These target percentages can be seen in Table 2. After calculating the percentages, the datasets then were ready to be run through the experimental code.

Both NEAT and ANN were allowed to run until one of three conditions was met: (1) the required correctness percentage from Table 2 , for both training and validation sets, was reached, (2) a predetermined number of generations had occurred, or (3) the allowed runtime on the server for the run had expired. The server that the experiments were run on was the COWBOY system at the Oklahoma State University High Performance Computing Center. NEAT was run starting with 0 hidden nodes and with 10 hidden nodes, and for a maximum of 1000 generations and 2000 generations. The ANN was run with 10 hidden nodes, a learning

rate of 0.1, and a maximum of 10,000,000 generations. The rest of the meta-parameters for NEAT were initialized using default values found in the p2test.ne parameter file that is distributed with the NEAT.[34]

Both AdaBoost with NEAT and AdaBoost with ANN were allowed to run until one of four conditions was met: (1) the required correctness percentage from Table 2, for both training and validation sets, was reached, (2) a predetermined number of generations had occurred, (3) the allowed runtime on the server for the run had expired, and (4) at least one weak hypothesis does not reach a correctness level greater than 50%. (As described in Section 2.4.4, each of the weak hypotheses created for use by AdaBoost must have correctness greater than 50%.)

3.2 Experiment

After all runs were completed the outcomes of NEAT and AdaBoost with NEAT were compared to see if the AdaBoost ensemble method could be used to improve the effectiveness of NEAT. These outcomes include the amount of evolution time each algorithm takes, the approximate number of evaluations needed, and the final accuracy of the hypothesis produced. The algorithms were evolved using the datasets chosen from the UCI machine learning repository, as listed in Section 3.1. The primary, preparatory changes to the data were switching the classification column from a character value to a numeric value, since NEAT requires numeric classification values. As previously stated in Section 3.1, the datasets were split apart for the creation of ten-fold cross-validation sets. For each of the original datasets, the data was divided randomly into ten partitions of approximately the same size, and the partitions were used to form ten different training set / validation set / testing set pairs. The training sets then were used to evolve neural networks, and both the training set and validation set were used to help determine when to stop training.

In all versions of the NEAT algorithm, the summation function used for all neurons was

the summation of inputs and bias, multiplied by their associated weights. Each neuron also used the log-sigmoid function (as shown in Table 1) as the activation function. As these experiments were run only on binary classification problems, the normally-expected activation function would either be the hard limit or the symmetrical hard limit function. Yet, since the backpropagation algorithm requires the activation function to be differentiable, these functions are not an option; but the log-sigmoid function can provide a good approximation of the hard limit function.

The results were compared to identify differences in the results as described below.

3.2.1 Decrease of Learning Time

The first experiment run was to test whether using AdaBoost with NEAT would decrease the learning time without a loss in accuracy of the results, as compared to NEAT. The learning time was calculated as the difference between the system time at the beginning of a training run and the system time at the completion; system time is an acceptable metric since the runs were executed on dedicated nodes of the Oklahoma State University High Performance Computing Center. As the experiments were performed using ten-fold cross-validation, the minimum time for each dataset will be used for comparison.

3.2.2 Increase of Accuracy

The second experiment was to test whether using AdaBoost with NEAT would decrease the number of function evaluations without a loss in accuracy of results, as compared to NEAT. The number of functions evaluations for ANN training can be calculated by taking the number of hidden nodes plus the number of output nodes, multiplied by the number of generations. This calculation is not as easy in NEAT because it can evolve hidden neurons, meaning that the number of hidden neurons possibly changes with each generation. The heuristic that was used to calculate function evaluations in NEAT and AdaBoost with NEAT took into account the number of hidden neurons in the last generation and used that in the

calculation for the number of function evaluations.

3.2.3 Overall Accuracy

The third experiment was to test whether AdaBoost with NEAT could provide a more accurate result than NEAT alone. This was tested by permitting the programs to run with a goal of 100% accuracy, and then to determine which of the two algorithms performed best when allowed to run to the maximum time allotted on the computer.

Chapter 4

Results

After all test runs were completed, those runs of NEAT and AdaBoost with NEAT having the same parameters (number of starting, hidden neurons and maximum number of generations) were compared. The runs were compared using the criteria given in Sections 3.2.1 and 3.2.2, i.e. total system time, and number of function evaluations, respectively.

However, if the accuracies of the runs of NEAT and AdaBoost with NEAT having the same parameters are different in a statistically significant way, one can come to a conclusion about which algorithm is more effective without using the criteria given in Sections 3.2.1 and 3.2.2.

Full results for all runs can be seen in Appendices A — D. A summarization of these results for all datasets - Spambase dataset, Adult dataset, Breast Cancer Wisconsin (Diagnostic) dataset, and Tic-Tac-Toe Endgame dataset - can be seen in Figures 5 — 8. Observations will be made concerning the results for each of the datasets, beginning with the Spambase dataset.

4.1 Results for the Spambase Dataset

Figure 9 provides a comparison of the accuracies resulting from the runs made for the Spambase dataset (hereafter referred to as SPAM). The paired t-test was run to help determine whether the accuracies for NEAT and AdaBoost with NEAT had a statistically significant difference, when run with the same parameters. For the paired t-test, a null

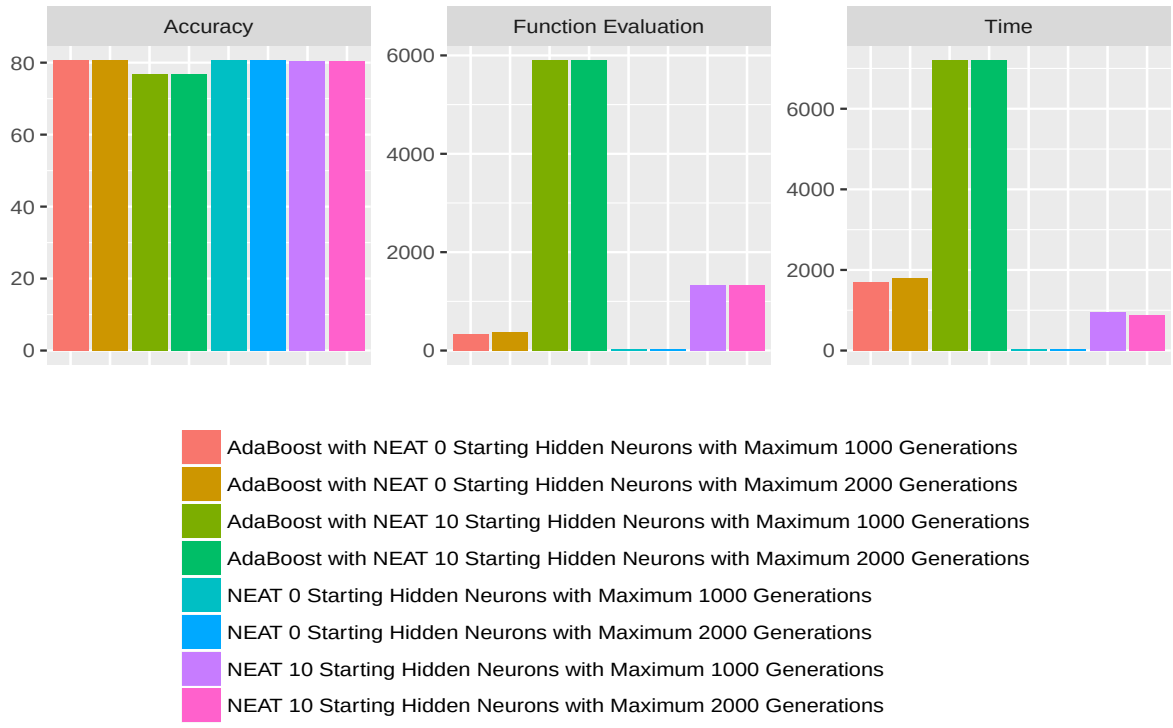

Figure 5: A Summary of the Adult Dataset Results

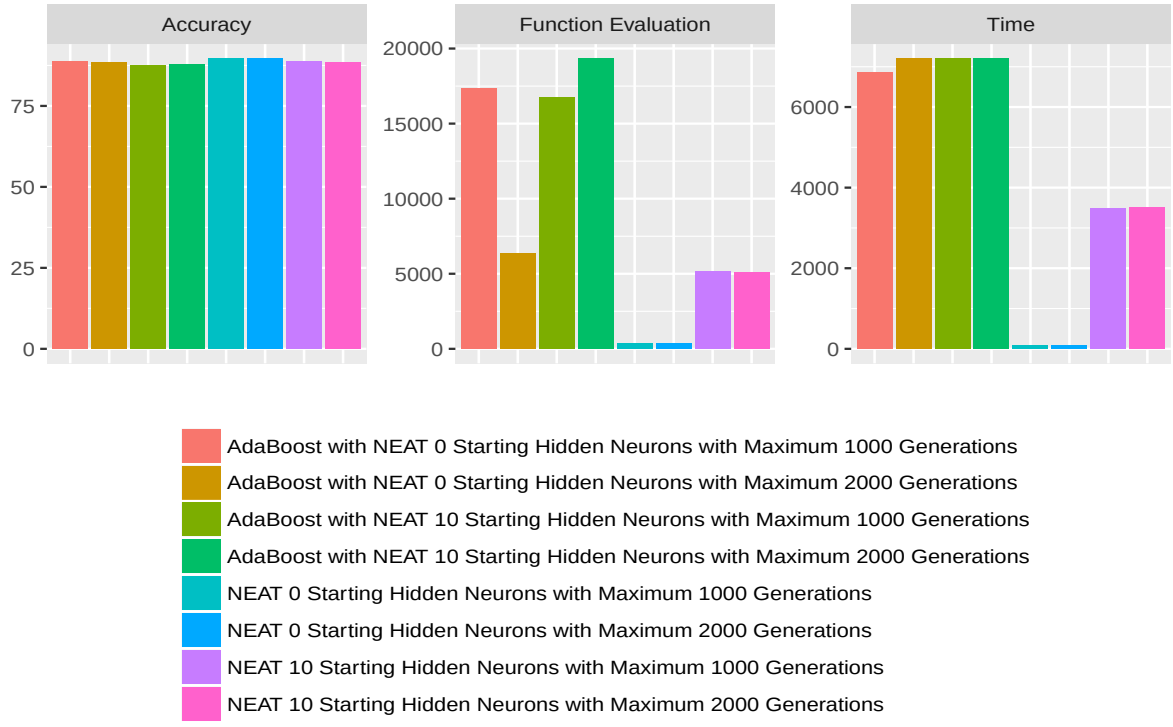

Figure 6: A Summary of the Spambase Dataset Results

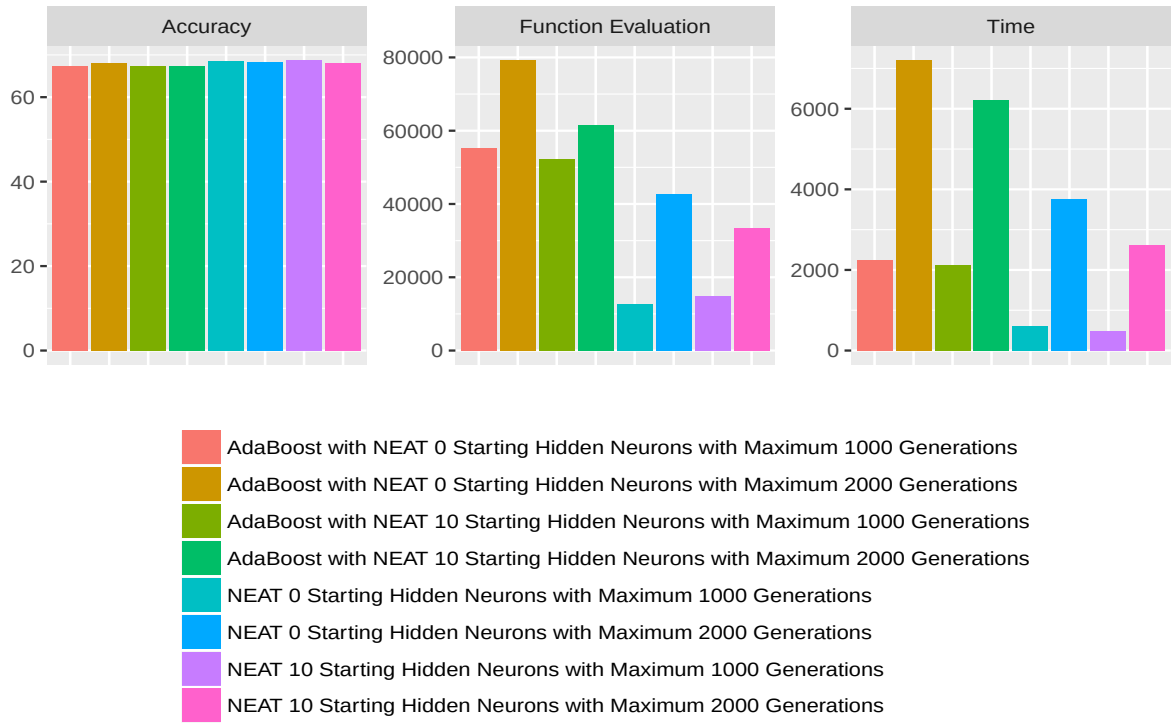

Figure 7: A Summary of the Tic-Tac-Toe Endgame Dataset Results

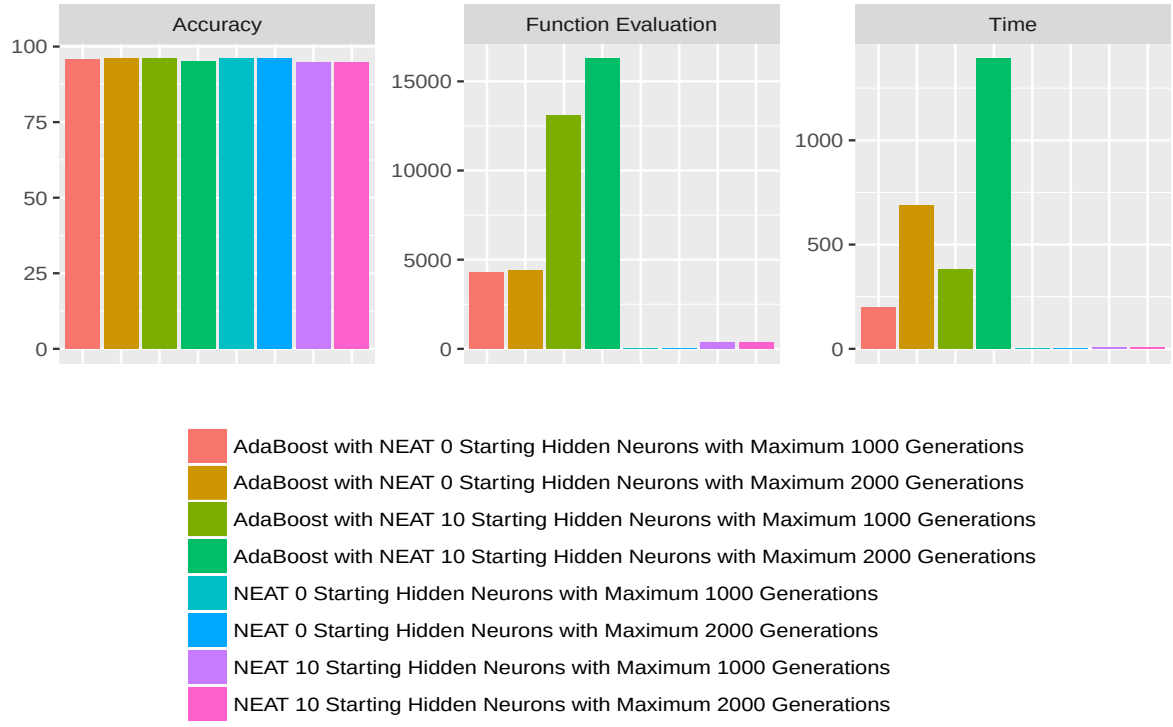

Figure 8: A Summary of the Wisconsin Breast Cancer (Diagnostic) Dataset Results

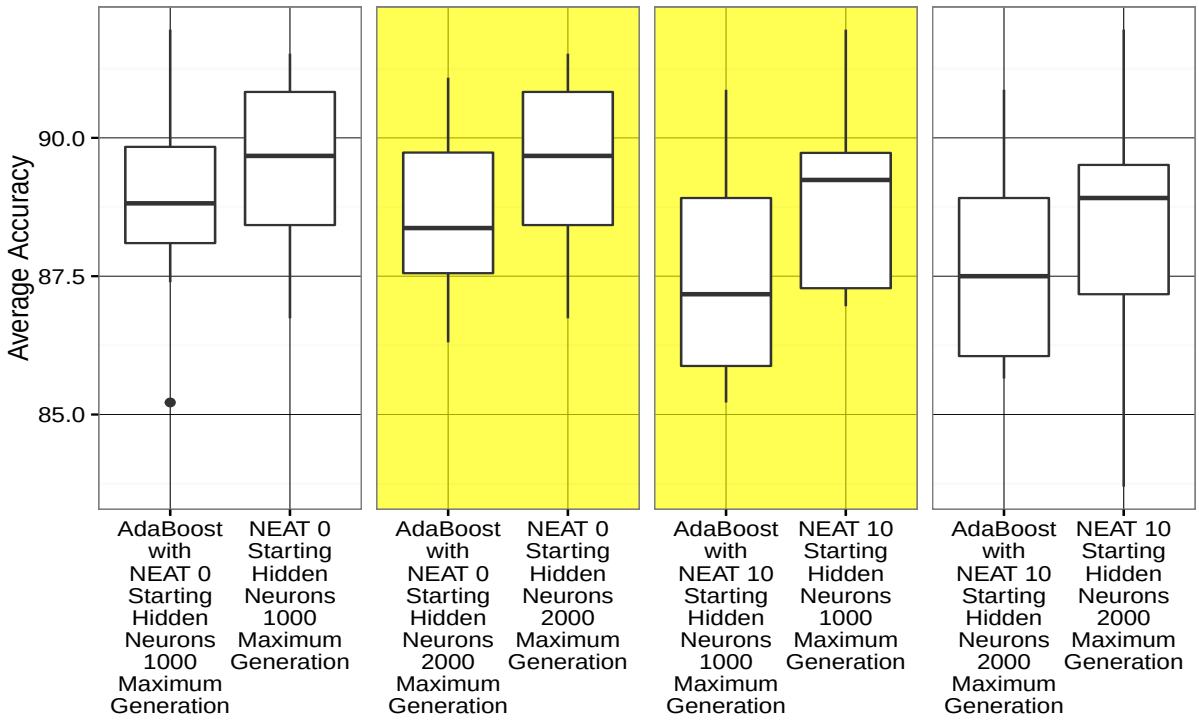

Figure 9: Average Accuracy of the Spambase Dataset

hypothesis (H_0) is that there is no statistically significant difference in accuracy of results between the two algorithms; the alternative hypothesis (H_α) is that the two algorithms do have a statistically significant difference. Normally, a p-value below 0.05 indicates that the null hypotheses can be rejected based upon the available data. As can be seen in Figure 9, two of the four sets of runs has a statistically significant difference in their accuracies, the highlighted comparisons indicate a statistically significant difference. This means that, for those runs, NEAT is a more effective choice over AdaBoost with NEAT as it provides a significantly more accurate result. The other three sets of runs were not statistically different, and other metrics had to be used to determine which algorithm was more effective.

Figures 10 and 11 provide a comparison of the training times and numbers of function evaluations for SPAM. Again, the t-test was run to help determine whether the results for NEAT and AdaBoost with NEAT had a statistically significant difference, when run with the same parameters. In this case a null hypothesis (H_0) is that there is no statistically

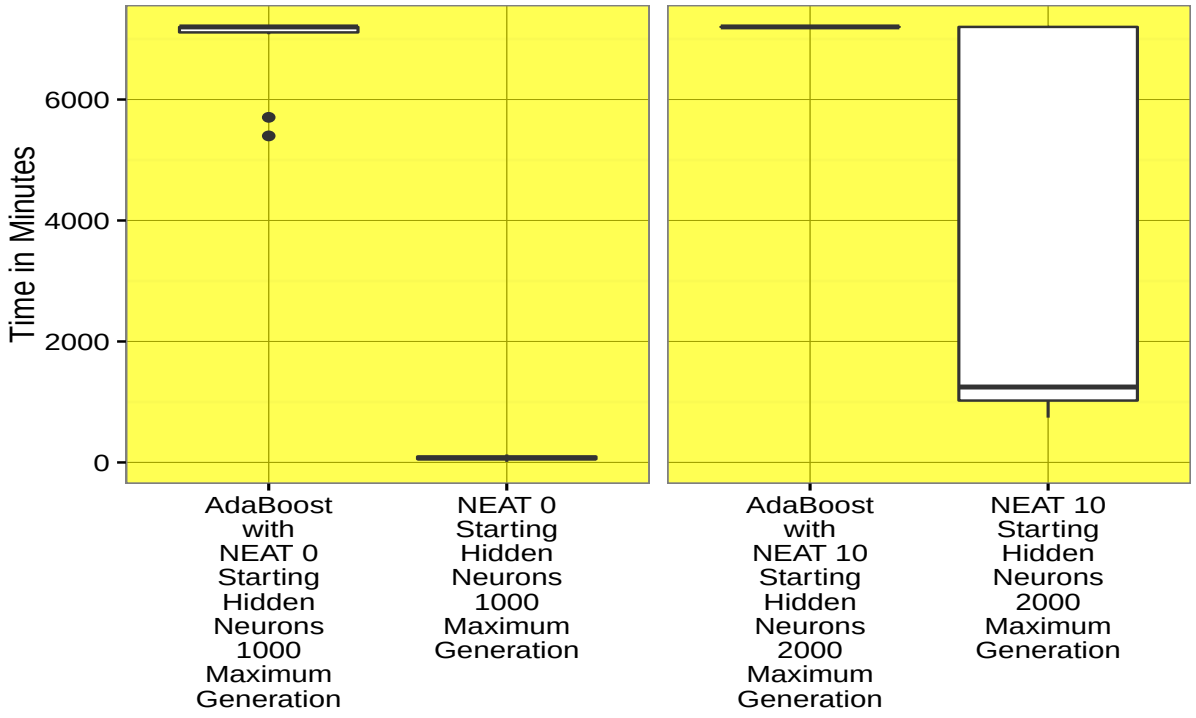

Figure 10: Time Comparisons in minutes of the Spambase Dataset

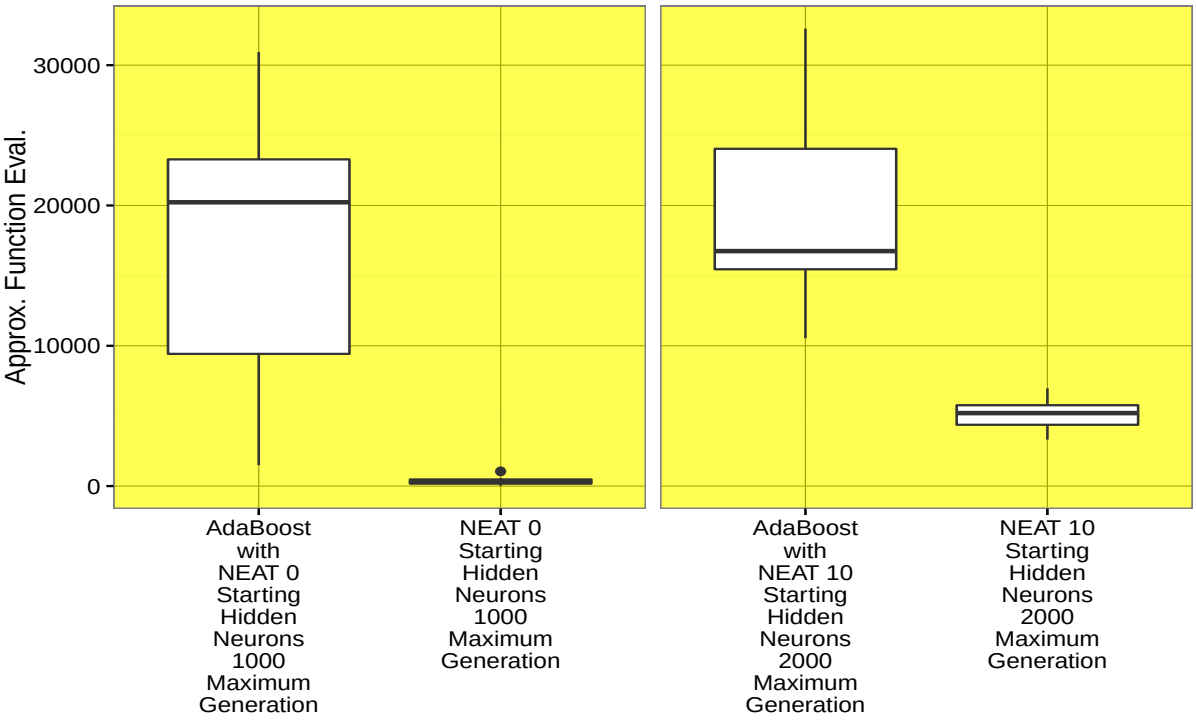

Figure 11: Average Number of Function Evaluations of the Spambase Dataset

significant difference in the results between the two algorithms; the alternative hypothesis (H_α) is that two algorithms do have a statistically significant difference.

As can be seen in Tables 10 and 11 – which contains only the SPAM runs with little statistical difference in accuracy – both sets of runs have a statistically significant difference in both training times and number of function evaluations. This means that, for those runs, NEAT is a more effective choice over AdaBoost with NEAT as it provides significantly less training time and significantly fewer function evaluations.

Thus, it can be concluded that for these tests involving the SPAM dataset, NEAT was more effective than AdaBoost with NEAT.

4.2 Results for the Adult Dataset

Figures 12, 13, and 14 provide a comparison of the accuracies, training time, and number of function evaluations resulting from the runs made for the Adult dataset (hereafter referred to as ADULT). As can be seen in Figure 12, two of the four sets of runs has a statistically significant difference in their accuracies. This means that, for those runs, NEAT is a more effective choice over AdaBoost with NEAT as it provides a significantly more accurate result. The other two sets of runs were not statistically different, and other metrics had to be used to determine which algorithm was more effective.

Figures 13 and 14 provide a comparison of the training times and numbers of function evaluations for ADULT. As can be seen, one of the two sets of runs have a statistically significant difference in training times and both sets have statistically significant differences in the number of function evaluations. The one set that doesn't have a statistically significant difference in training time, it is very close with a P-value of 0.051651942. This means that, for those runs, NEAT is a more effective choice over AdaBoost with NEAT as it provides less training time and significantly fewer function evaluations.

Thus, it can be concluded that for these tests involving the ADULT dataset, NEAT was

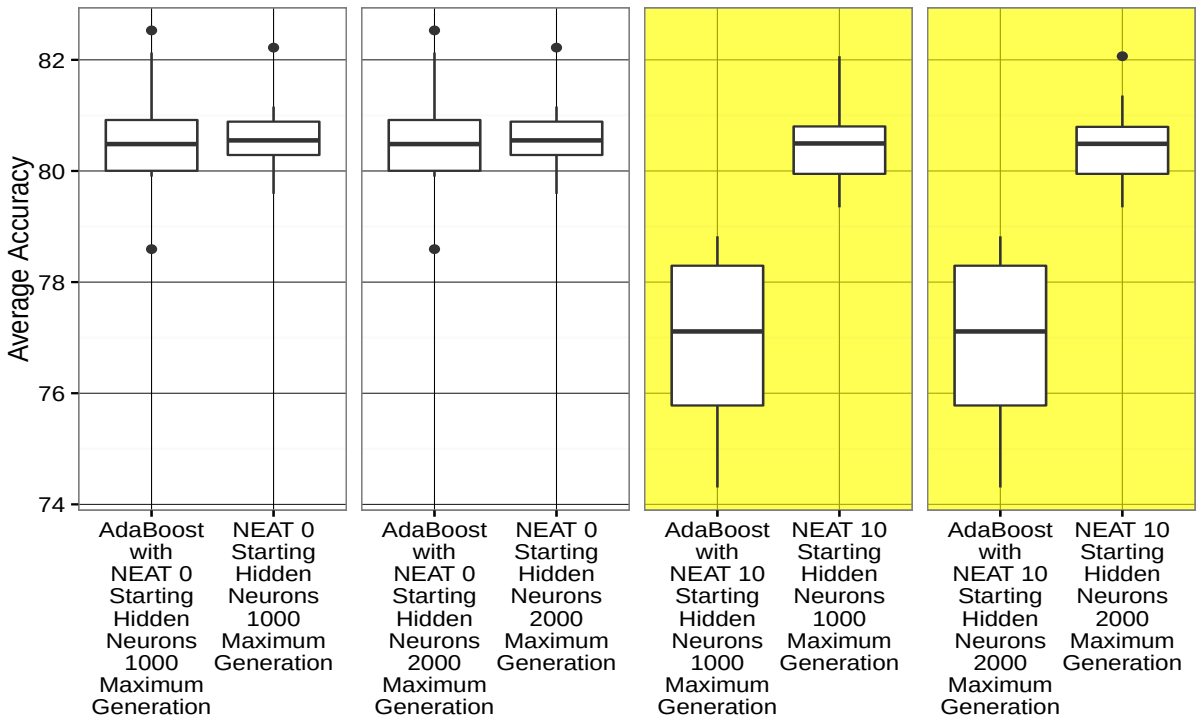

Figure 12: Accuracy of the Adult Dataset

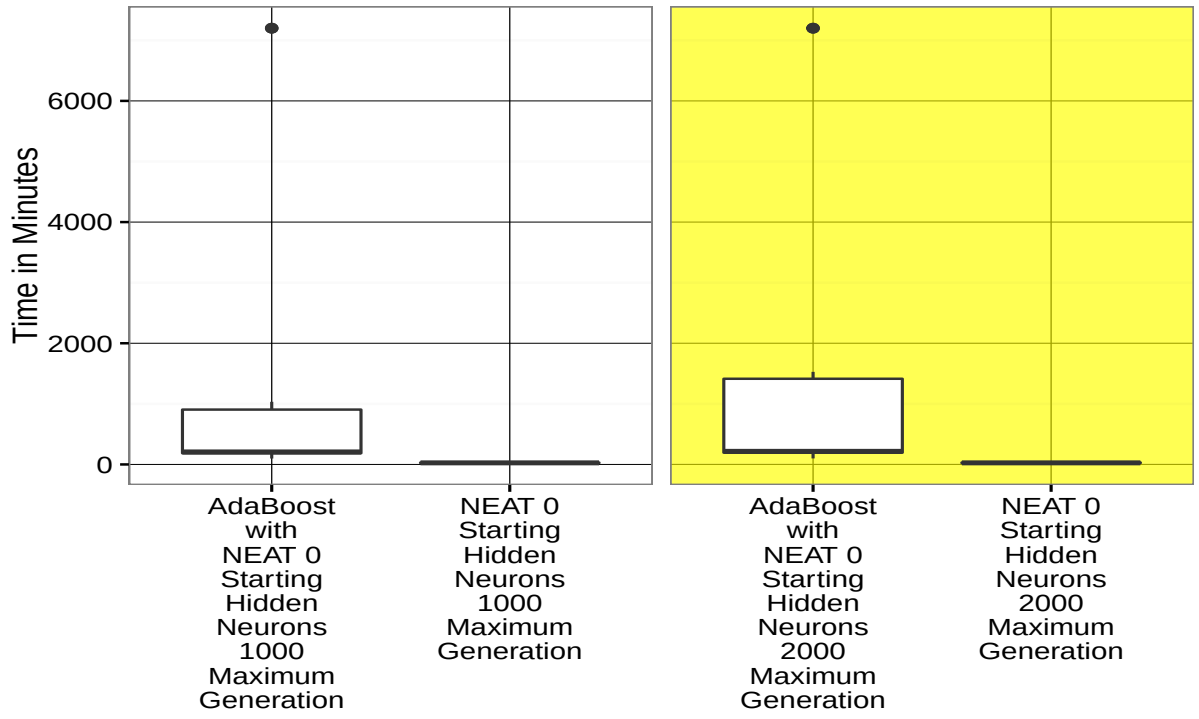

Figure 13: Time Comparisons in minutes of the Adult Dataset

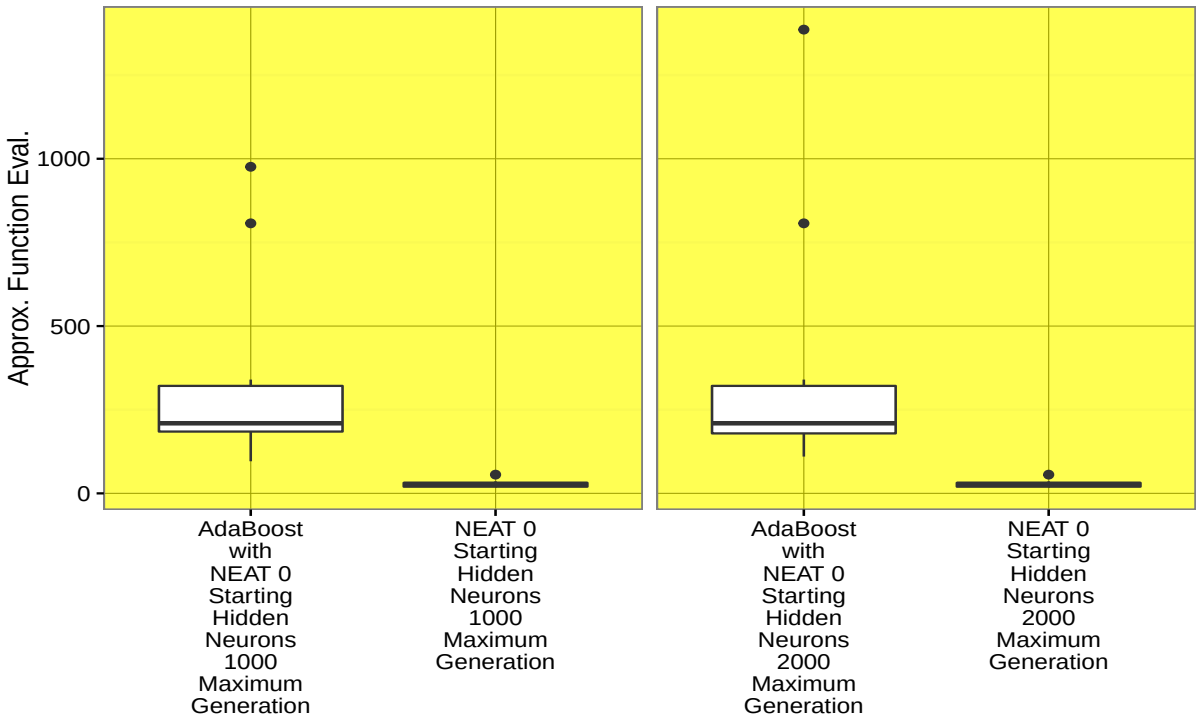

Figure 14: Average Number of Function Evaluations of the Adult Dataset

more effective than AdaBoost with NEAT.

4.3 Results for the Tic-Tac-Toe Endgame Dataset

Figures 15, 16, and 17 provide a comparison of the accuracies, training time, and number of function evaluations resulting from the runs made for the Tic-Tac-Toe Endgame dataset (hereafter referred to as TTT). As can be seen in Figure 15, none of the four sets of runs has a statistically significant difference in their accuracies. This means that, for those runs, none were statistically different, and other metrics had to be used to determine which algorithm was more effective.

Figures 16 and 17 provide a comparison of the training times and numbers of function evaluations for TTT. As can be seen, all but one sets of runs have a statistically significant difference in both training times and number of function evaluations. The one set that does

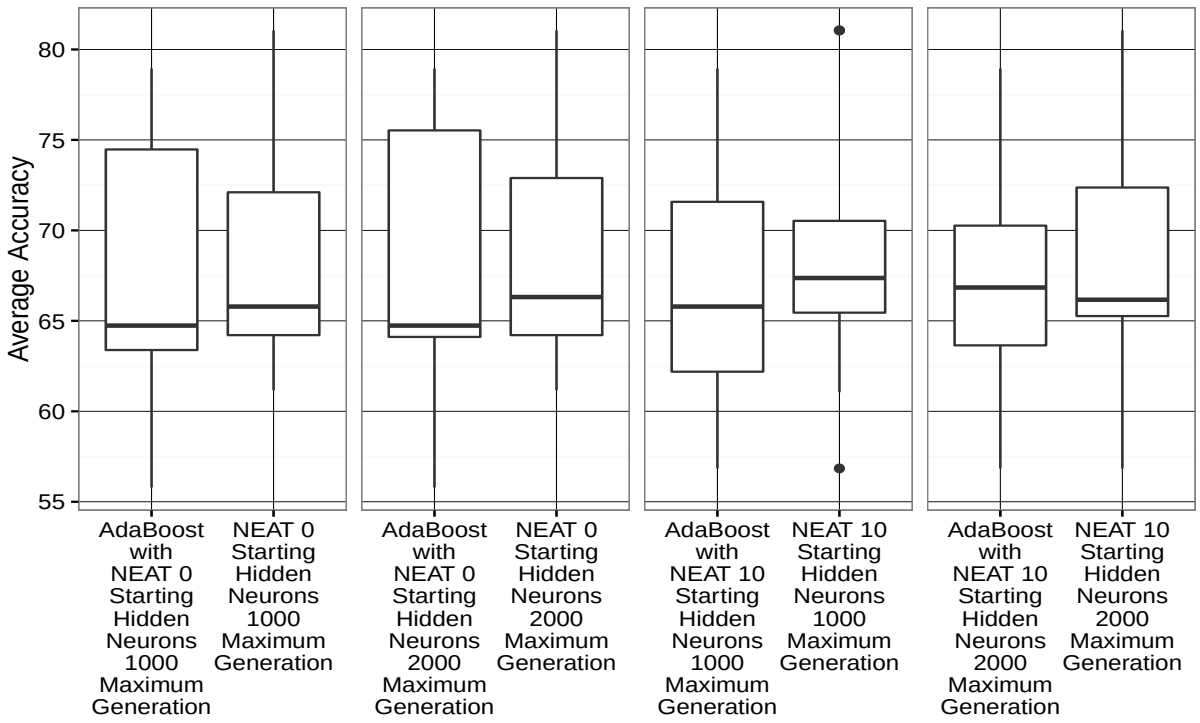

Figure 15: Accuracy of the Tic-Tac-Toe Endgame Dataset

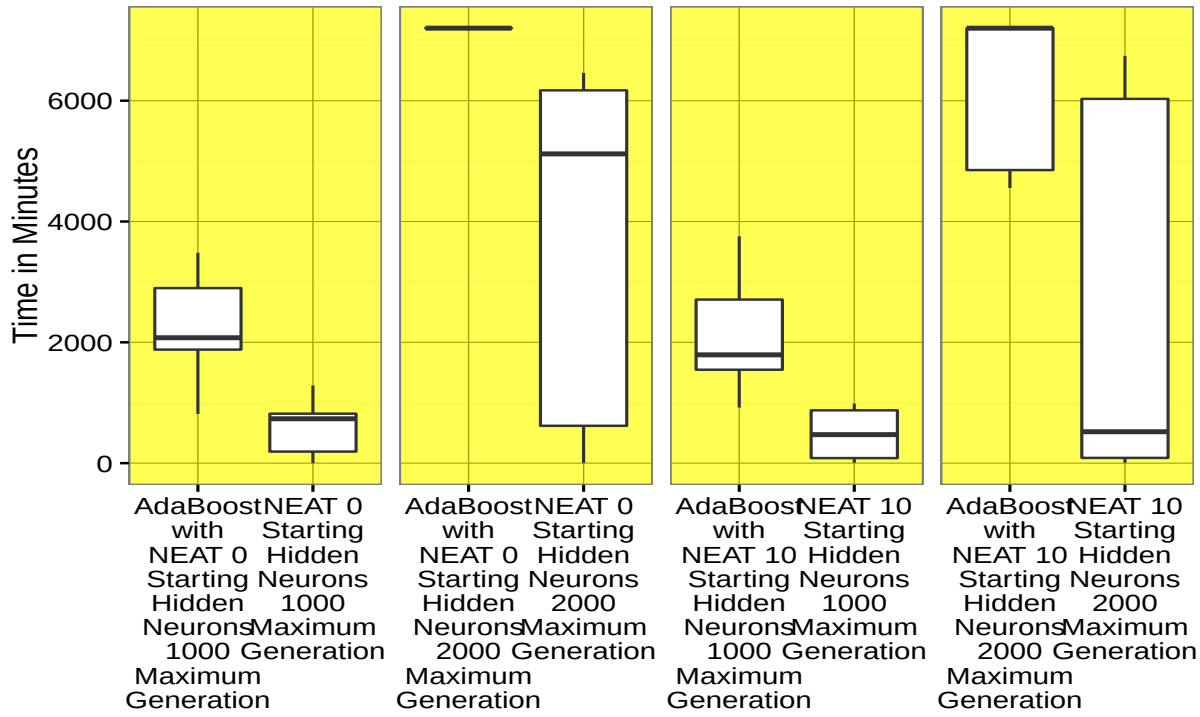

Figure 16: Time Comparisons in minutes of the Tic-Tac-Toe Endgame Dataset

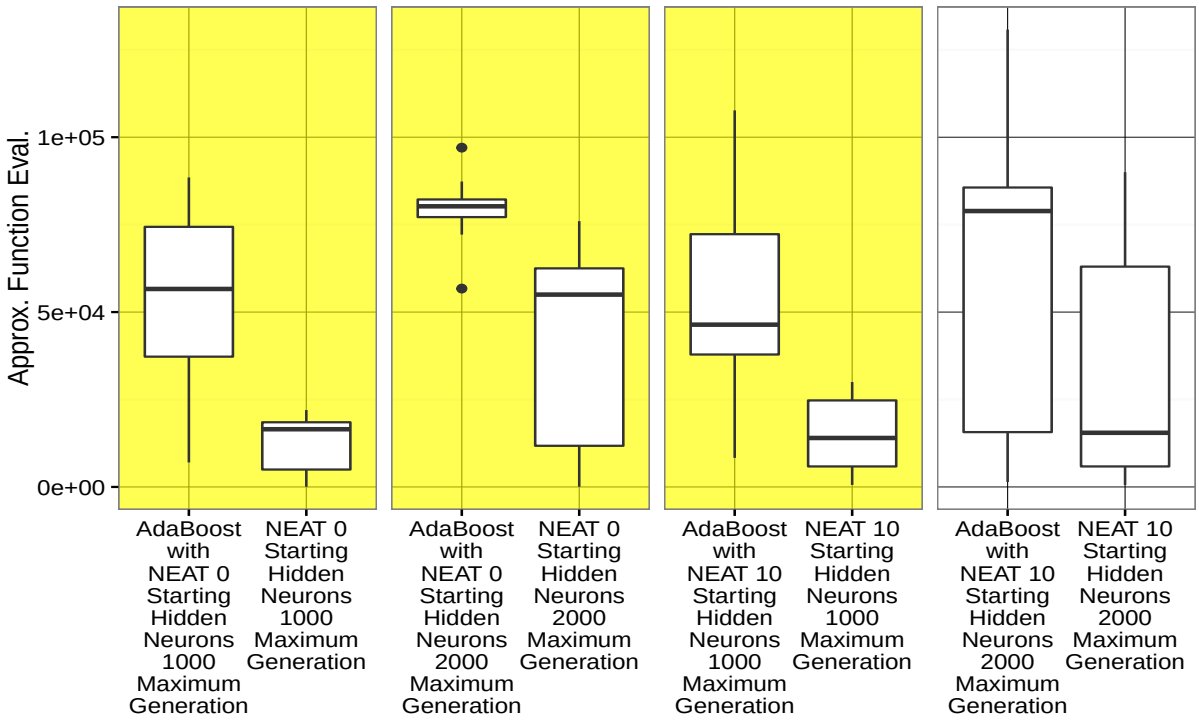

Figure 17: Average Number of Function Evaluations of the Tic-Tac-Toe Endgame Dataset

not have a statistically significant difference is the NEAT and AdaBoost with NEAT starting with ten hidden neurons and a maximum of 2000 generations when comparing the number of function evaluations. While this set is not statistically significant different, there is a major difference in the average number of function evaluations. This means that, for those runs, NEAT is a more effective choice over AdaBoost with NEAT as it provides significantly less training time and fewer function evaluations.

Thus, it can be concluded that for these tests involving the TTT dataset, NEAT was more effective than AdaBoost with NEAT.

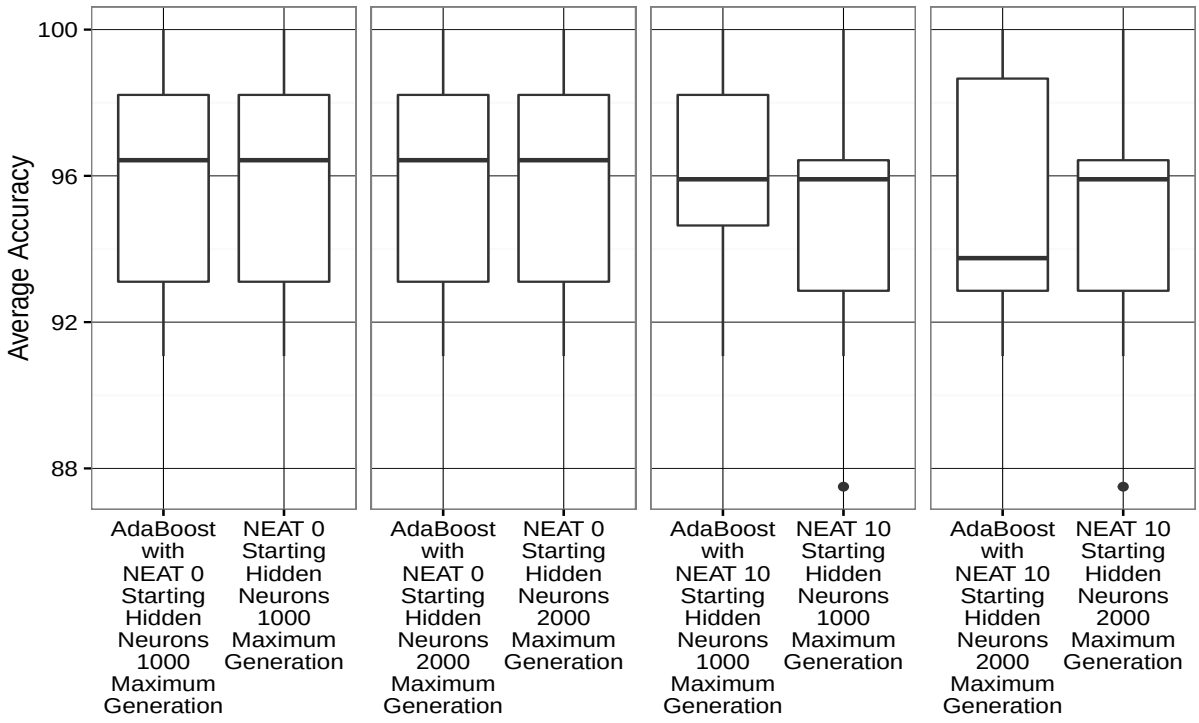

Figure 18: Accuracy of the Wisconsin Breast Cancer (Diagnostic) Dataset

4.4 Results for the Breast Cancer Wisconsin (Diagnostic) Dataset

Figures 18, 19, and 20 provide a comparison of the accuracies, training time, and number of function evaluations resulting from the runs made for the Wisconsin Breast Cancer (Diagnostic) dataset (hereafter referred to as WDBC). As can be seen in Figure 18, none of the four sets of runs has a statistically significant difference in their accuracies. This means that, for those runs, none were statistically different, and other metrics had to be used to determine which algorithm was more effective.

Figures 19 and 20 provide a comparison of the training times and numbers of function evaluations for WDBC. As can be seen, each of the different comparisons have three of the four sets of runs have a statistically significant difference in both training times and number of function evaluations. The two sets that are not statistically significant differences has NEAT

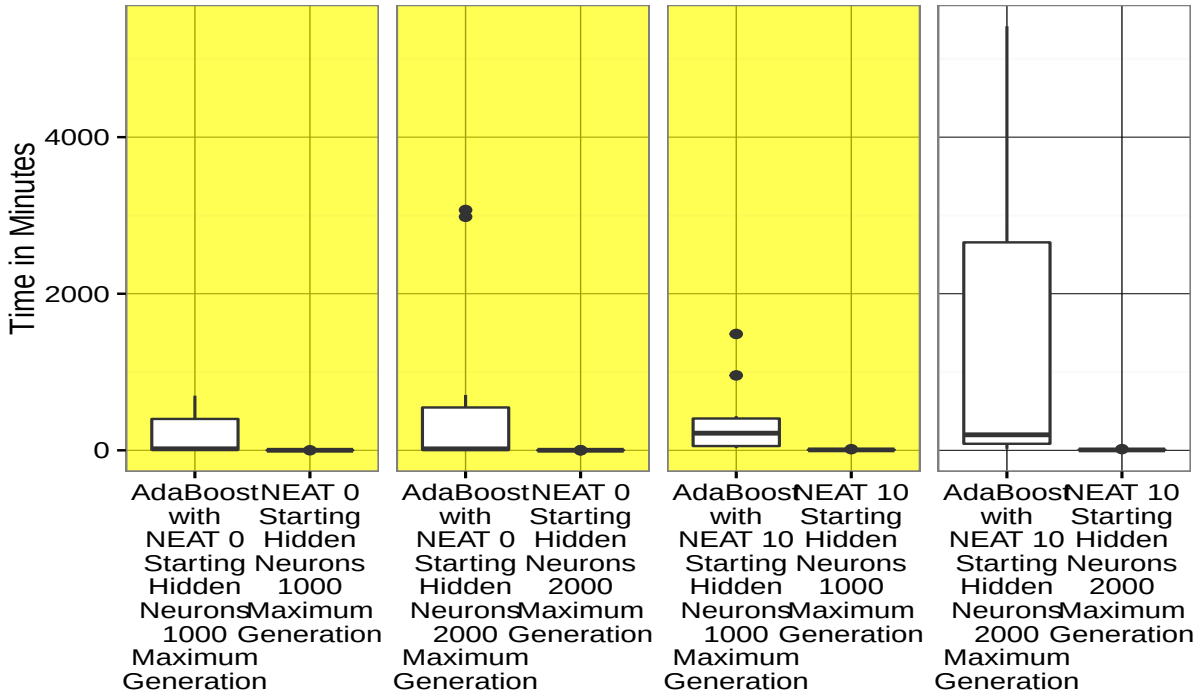

Figure 19: Time Comparisons in minutes of the Wisconsin Breast Cancer (Diagnostic) Dataset

having a smaller average training time and fewer average number of function evaluations. This means that, for those runs, NEAT is a more effective choice over AdaBoost with NEAT as it provides less training time and fewer function evaluations.

Thus, it can be concluded that for these tests involving the WDBC dataset, NEAT was more effective than AdaBoost with NEAT.

4.5 Accuracy Runs

Tables 21 — 24 provide a comparison of the accuracies of NEAT and AdaBoost with NEAT when trained toward a goal of 100% accuracy on the validation set for each of the four different datasets. As can be seen, NEAT provides a more accurate result for all four datasets, and especially for the ADULT and SPAM datasets, where the differences are statistically significant according to paired T-Tests.

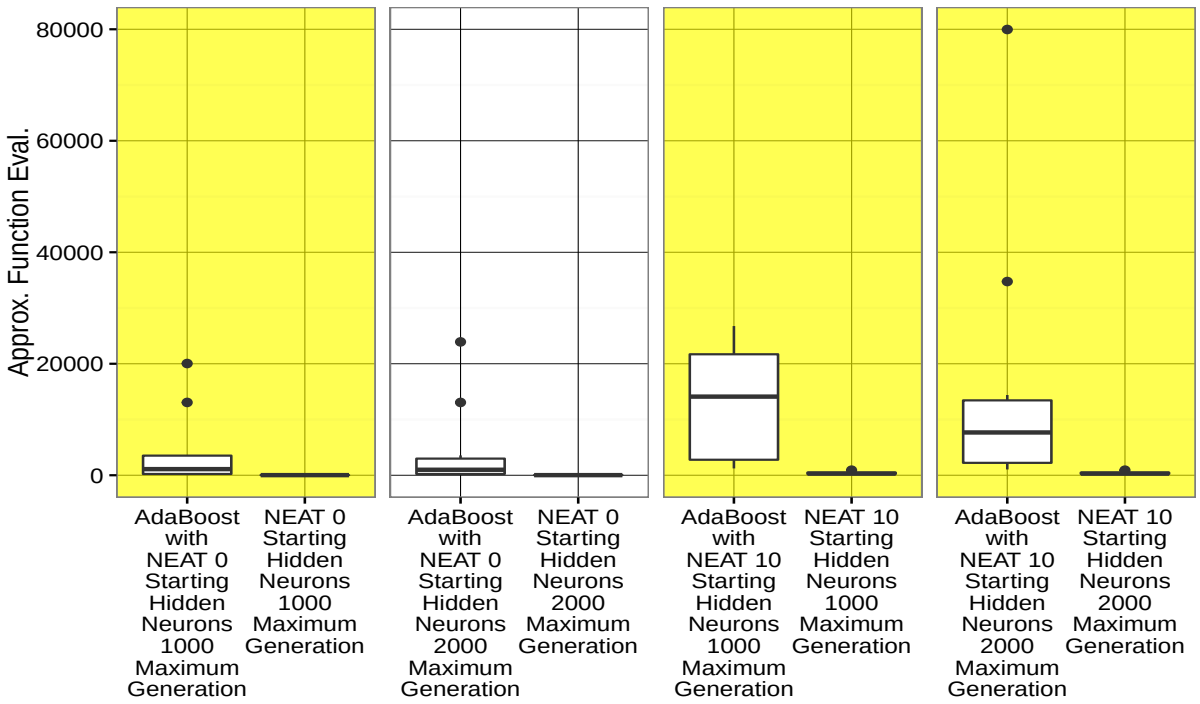

Figure 20: Average Number of Function Evaluations of the Wisconsin Breast Cancer (Diagnostic) Dataset

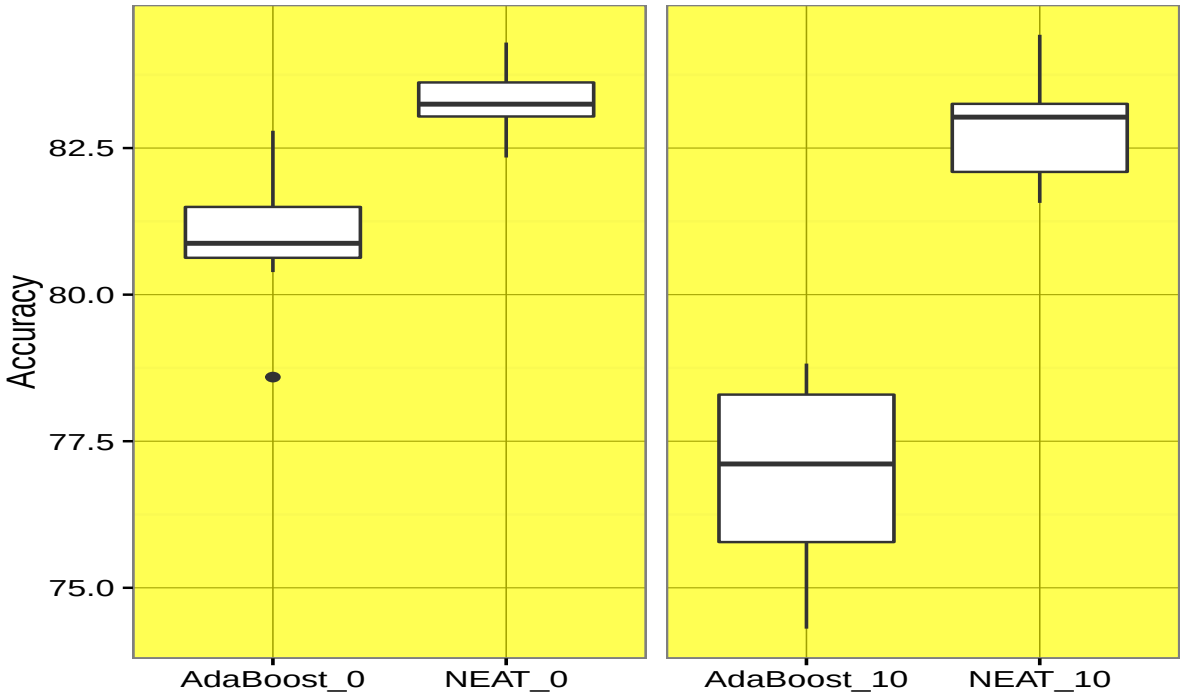

Figure 21: Accuracy of ADULT Dataset Max Runtime

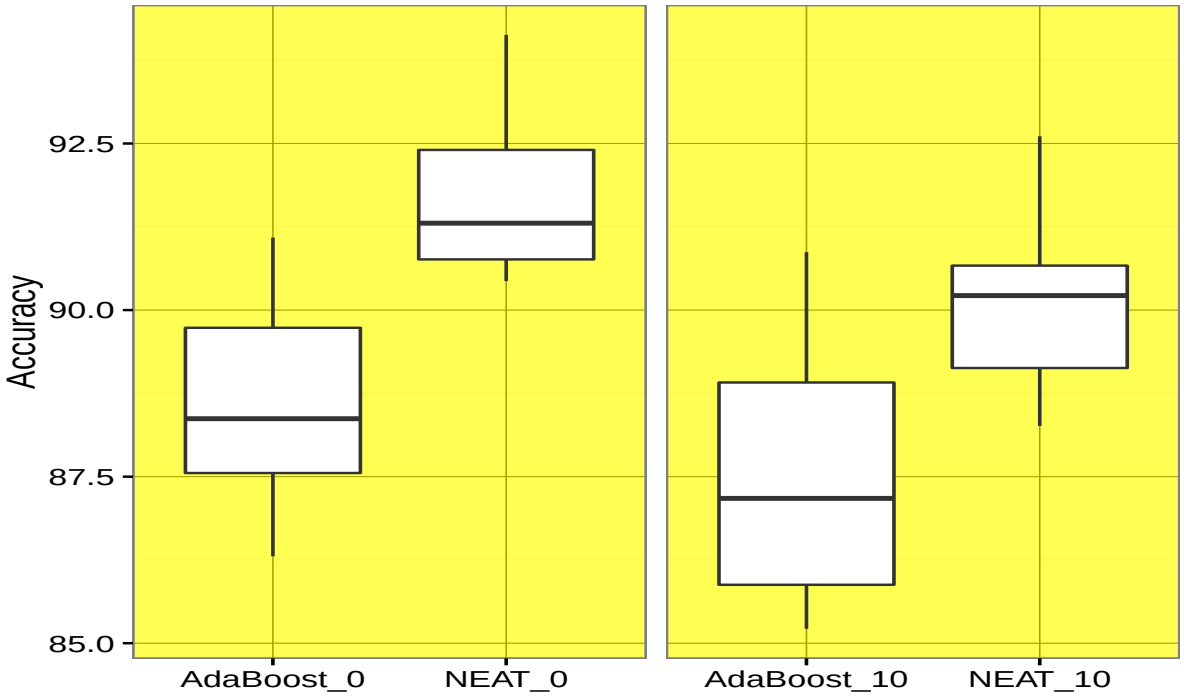

Figure 22: Accuracy of SPAM Dataset Max Runtime

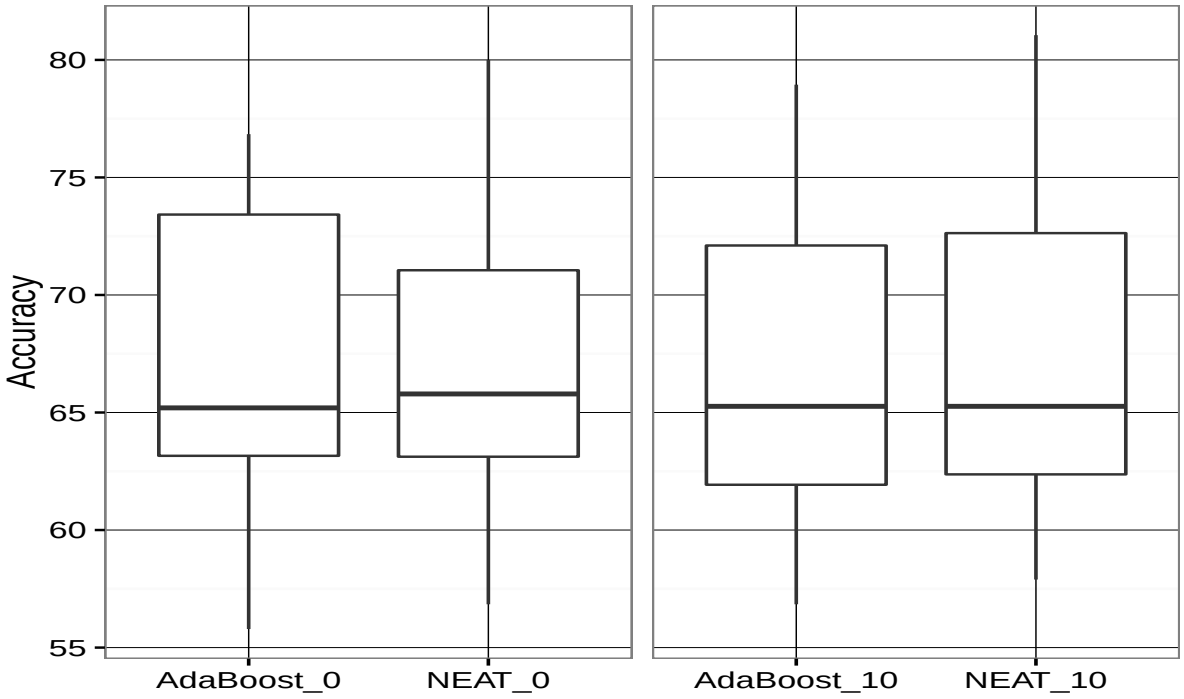

Figure 23: Accuracy of TTT Dataset Max Runtime

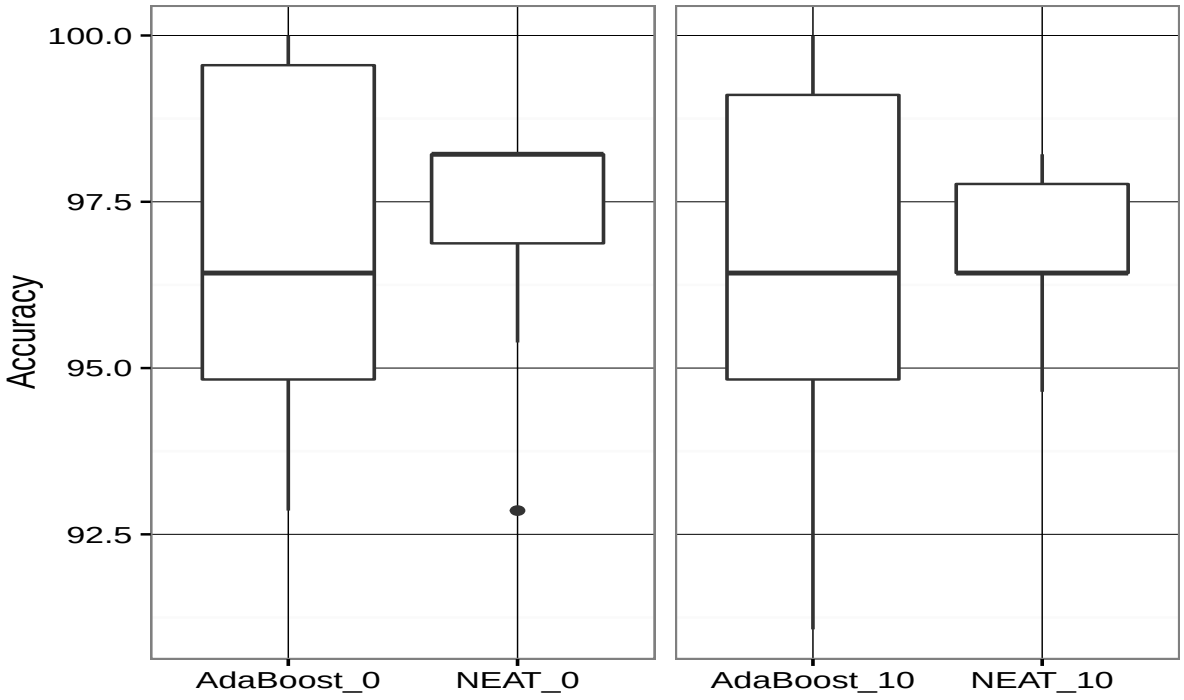

Figure 24: Accuracy of WDBC Dataset Max Runtime

Thus, it can be concluded that when trying to determining which algorithm is better for accuracies, NEAT was more effective than AdaBoost with NEAT for these cases.

4.6 Overall Observation

The results of these tests indicated that, for all of the datasets used, there were no instances in which AdaBoost with NEAT was more effective than NEAT alone. More specifically, AdaBoost with NEAT added to the computational time and the number of function evaluations as compared to NEAT.

Chapter 5

Conclusion and Future Work

There are many different classification algorithms that use ANNs; one of these is the NEAT algorithm.[4, 5, 6, 7] This research tested the use of an ensemble algorithm (AdaBoost [1, 8] with NEAT) in an attempt to find a way to improve on the performance of the NEAT algorithm used alone. Four datasets were chosen from the UCI machine learning repository for use in these experiments: Adult Salary dataset, Spambase dataset (information about spam and ham emails), Breast Cancer Wisconsin (Diagnostic) dataset, and Tic-Tac-Toe Endgame dataset.[9] The first three appear in the literature regarding classification algorithms. The final dataset was chosen because it looked like a good candidate for these experiments.

An implementation of the NEAT algorithm was retrieved from the University of Texas Neural Networks Research Group.[34] An implementation of the AdaBoost algorithm was developed, and this implementation used the University of Texas's NEAT implementation. The datasets were randomly partitioned into training sets, validation sets, and testing sets using ten-fold cross-validation techniques, and the data was normalized. Runs were made for both NEAT and AdaBoost with NEAT using the partitioned data. The runs were executed on the Cowboy cluster of the Oklahoma State University High Performance Computing Center.

After all runs were completed, the results were gathered for analysis. The statistical paired t-test was performed to compare the results of the runs of NEAT to those of AdaBoost with NEAT. Given the importance of accurate classification, this is the first area to which

the statistical test was applied. Some results were found to have statistically significant differences in their accuracies, clearly indicating which algorithm was more effective in those cases. For other cases, where there were no significant differences in accuracy, the statistical test were applied to analyze differences in training times and numbers of function evaluations between the two algorithms.

The results of the tests indicated that, for the datasets used in these experiments, there were no instances in which AdaBoost with NEAT was more effective than NEAT alone. More specifically, AdaBoost with NEAT added to the computational time and the number of function evaluations as compared to NEAT. These experiments indicate that using AdaBoost in ensemble with NEAT may not be beneficial.

In later experiments, both algorithms were permitted to run towards 100% accuracy on the validation dataset. As seen in tables 21 - 24, NEAT provides more accurate results for all four test datasets, and significantly better results in two of the datasets.

One reason for this may be that NEAT is designed to evolve the topology of neural networks, so that they can learn to classify the data with which they are presented. AdaBoost with NEAT attempts to focus the learning process on the more difficult classifications, and in doing so, it causes NEAT to make a greater number of function evaluations, there by increasing training time.

There are other questions that could be asked and tested in the future concerning the use of AdaBoost with NEAT. The experiments described in this paper use the AdaBoost.M2 algorithm, which was designed specifically for neural networks; other implementations of AdaBoost could be used to see if they produce different or better results. AdaBoost (or a derivation of AdaBoost) could be replaced by other ensemble methods to see whether better results could be achieved. Further, it might be worthwhile to test AdaBoost with non-binary classification datasets.

Bibliography

[1] H. Schwenk and Y. Bengio, "Boosting neural networks," *Neural Comput.*, vol. 12, no. 8, pp. 1869–1887, 2000.

[2] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[3] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, second ed., 2003.

[4] K. O. Stanley, *Efficient Evolution of Neural Networks Through Complexification*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2004.

[5] K. O. Stanley and R. Miikkulainen, "Continual coevolution through complexification," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)* (W. B. Langdon, E. Cantu-Paz, K. E. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, and A. C. Schultz, eds.), (San Francisco), p. 8, Morgan Kaufmann, 2002.

[6] K. O. Stanley and R. Miikkulainen, "Efficient evolution of neural network topologies," in *Proceedings of the Genetic and Evolutionary Computation Conference* (W. B. Langdon, E. Cantu-Paz, K. E. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, and A. C. Schultz, eds.), (Piscataway, NJ), pp. 1757–1762, San Francisco, CA: Morgan Kaufmann, 2002.

[7] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[8] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," in *EuroCOLT '95: Proceedings of the Second European Conference on Computational Learning Theory*, (London, UK), pp. 23–37, Springer-Verlag, 1995.

[9] M. Lichman, "UCI machine learning repository," 2013.

[10] M. Hagan, H. Demuth, and M. Beale, *Neural Network Design*. Electrical Engineering Series, Pws Pub., 1996.

[11] E. Bonabeau, M. Dorigo, and G. Theraulaz, *From Natural to Artificial Swarm Intelligence*. Oxford University Press, 1999.

[12] E. Turban and L. Frenzel, *Expert systems and applied artificial intelligence*. The Macmillan series in information technology, Macmillan Pub. Co., 1992.

[13] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359 – 366, 1989.

[14] M. L. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge Mass.: MIT Press, expanded ed., 1988.

[15] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological Review*, vol. 65, no. 6, pp. 386 – 408, 1958.

[16] P. Haley and D. Soloway, "Extrapolation limitations of multilayer feedforward neural networks," in *International Joint Conference on Neural Networks*, vol. 4, pp. 25–30 vol.4, 1992.

[17] M. Kearns and U. Vazirani, *An introduction to computational learning theory*. MIT Press, 1994.

[18] A. Kent and J. Williams, *Encyclopedia of Computer Science and Technology: Volume 45 - Supplement 30*. Encyclopedia of Computer Science Series, Taylor & Francis, 2002.

[19] M. Wiering and H. van Hasselt, "Ensemble algorithms in reinforcement learning," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 38, no. 4, pp. 930–936, 2008.

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

[21] M. R. Chernick and R. A. LaBudde, *An Introduction to Bootstrap Methods with Applications to R*. Wiley Publishing, 1st ed., 2011.

[22] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. New York: Chapman & Hall, 1993.

[23] N. Littlestone and M. Warmuth, "The weighted majority algorithm," *Information and Computation*, vol. 108, no. 2, pp. 212 – 261, 1994.

[24] S. A. Goldman and M. K. Warmuth, "Learning binary relations using weighted majority voting," in *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, COLT '93, (New York, NY, USA), pp. 453–462, ACM, 1993.

[25] K. De Jong, "Learning with genetic algorithms: An overview," *Machine learning*, vol. 3, no. 2-3, pp. 121–138, 1988.

[26] T. Bäck, *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford, UK: Oxford University Press, 1996.

[27] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Accelerated neural evolution through cooperatively coevolved synapses," *J. Mach. Learn. Res.*, vol. 9, pp. 937–965, June 2008.

[28] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, (San Francisco, CA, USA), pp. 1137–1143, Morgan Kaufmann Publishers Inc., 1995.

[29] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2009.

[30] W. Wolberg and O. Mangasarian, "Multisurface method of pattern separation for medical diagnosis applied to breast cytology,," in *Proceedings of the National Academy of Sciences*, pp. 9193–9196, Dec 1990.

[31] M. G. Mini, "Neural network based classification of digitized mammograms," in *Proceedings of the Second Kuwait Conference on e-Services and e-Systems*, KCESS '11, (New York, NY, USA), pp. 2:1–2:5, ACM, 2011.

[32] J. Farkas, "Document classification and recurrent neural networks," in *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '95, pp. 21–, IBM Press, 1995.

[33] Y. Jiang, D. Wang, R. Liu, and Z. Feng, "Binaural classification for reverberant speech segregation using deep neural networks," *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, vol. 22, pp. 2112–2121, Dec. 2014.

[34] "Nnrg software - neat c++." `http://nn.cs.utexas.edu/?neat-c`. Accessed: 2016-01-05.

[35] S. Esmeir and S. Markovitch, "Lookahead-based algorithms for anytime induction of decision trees," in *ICML*, 2004.

[36] B. Hamers and J. A. K. Suykens, "Coupled transductive ensemble learning of kernel models," *Journal of Machine Learning Research 1*, 2003. bart.hamers@esat.kuleuven.ac.be.

Appendix A

Adult Dataset Percent Correct Comparison Graphs

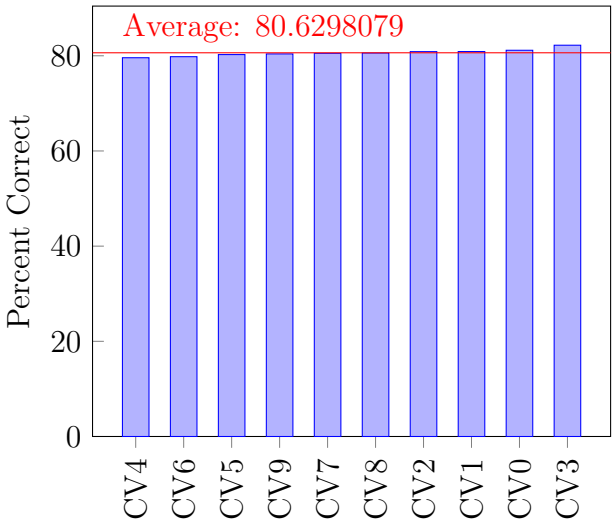

Figure 25: Percent Correct NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

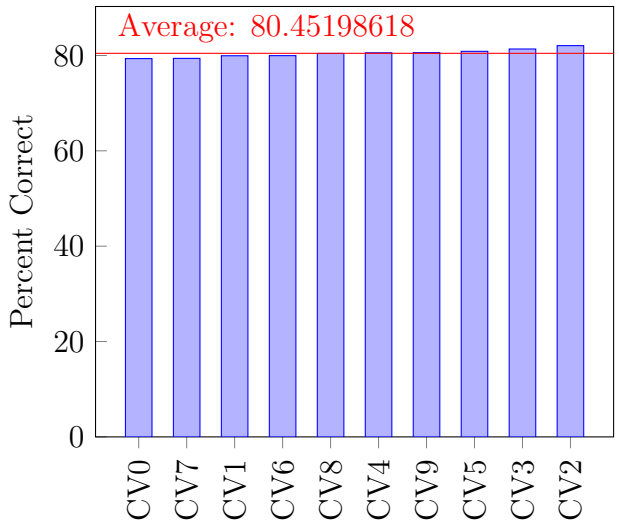

Figure 26: Percent Correct NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

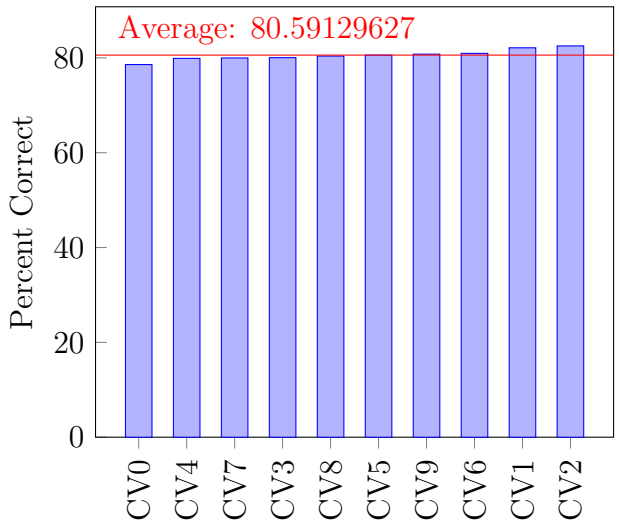

Figure 27: Percent Correct AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

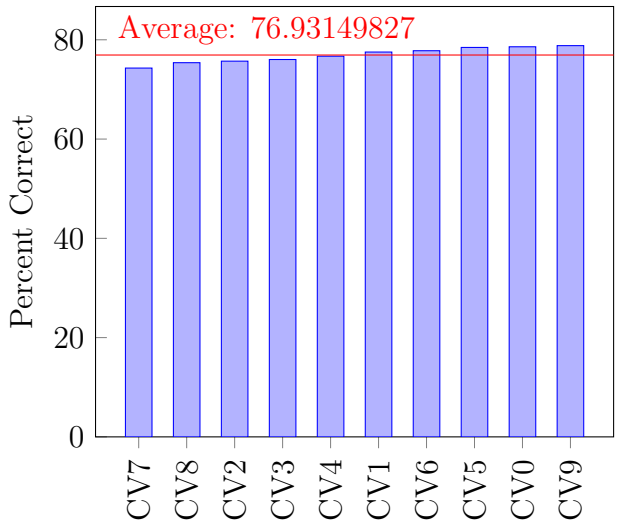

Figure 28: Percent Correct AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

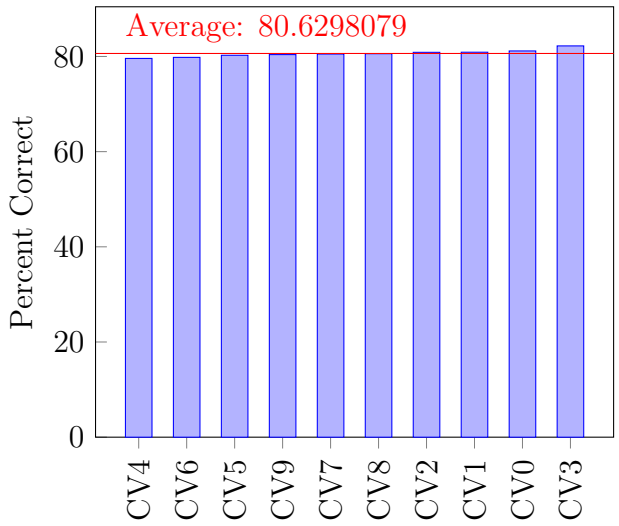

Figure 29: Percent Correct NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

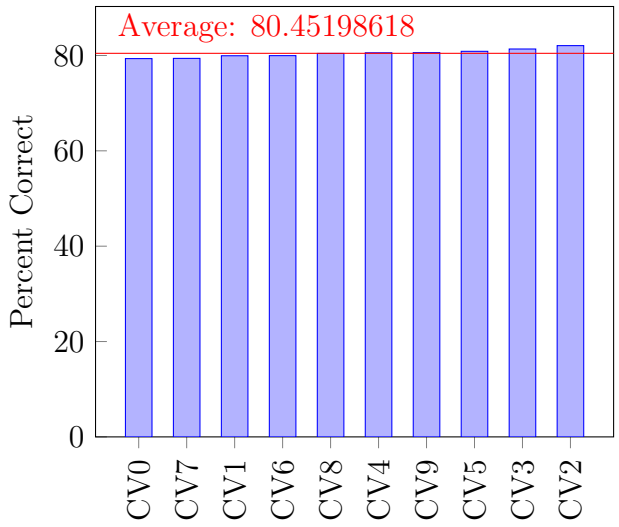

Figure 30: Percent Correct NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

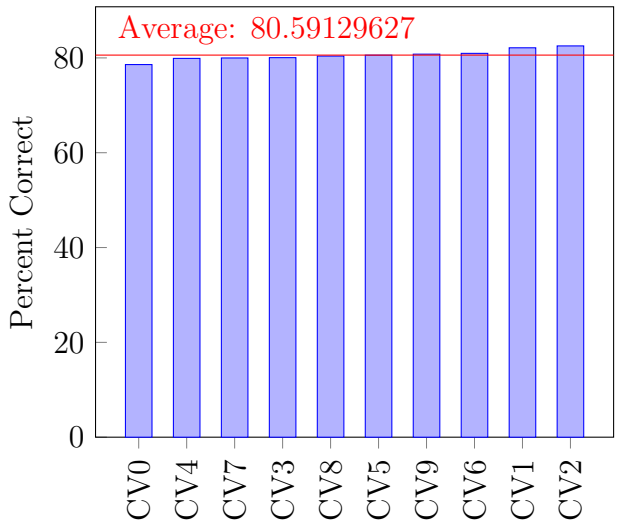

Figure 31: Percent Correct AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

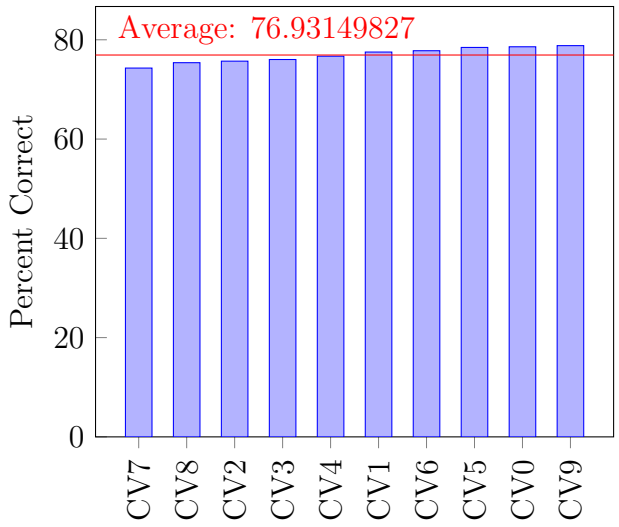

Figure 32: Percent Correct AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

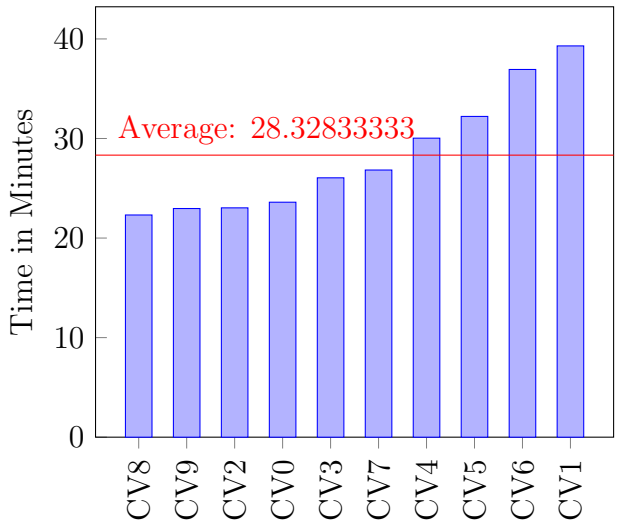

Figure 33: Time Spent Learning Using NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

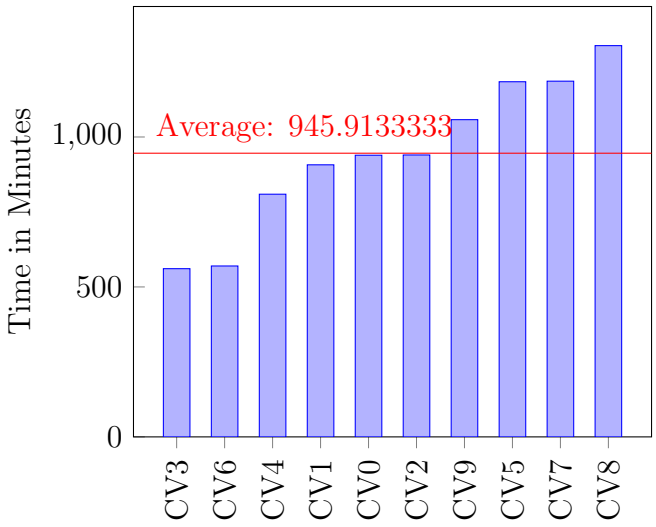

Figure 34: Time Spent Learning Using NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

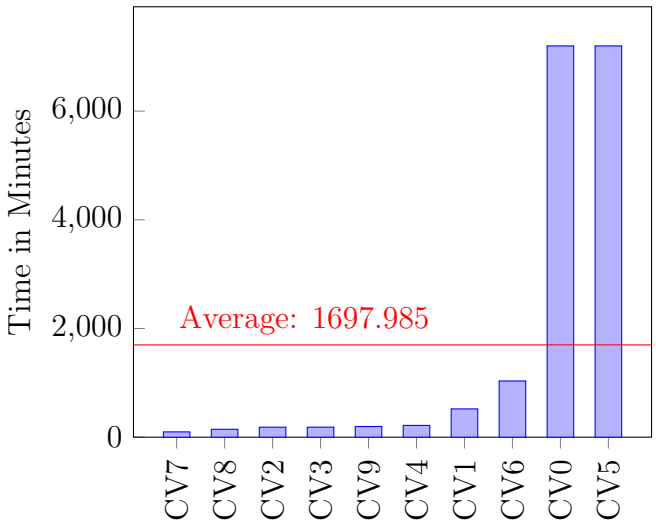

Figure 35: Time Spent Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

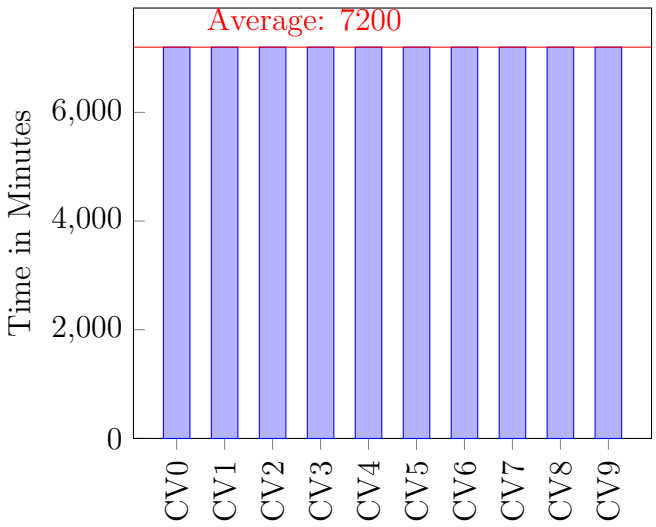

Figure 36: Time Spent Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

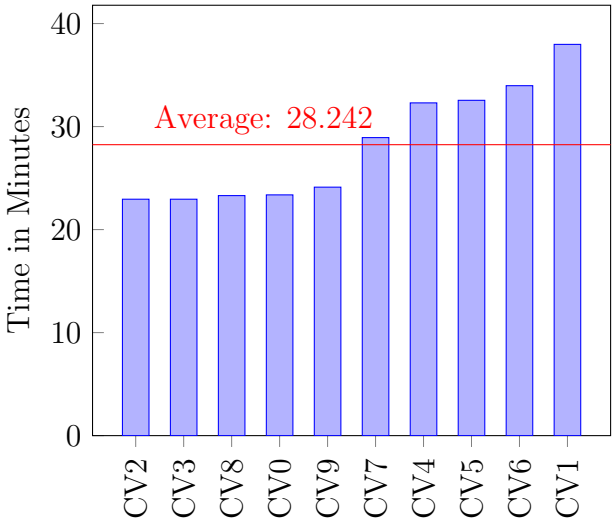

Figure 37: Time Spent Learning Using NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

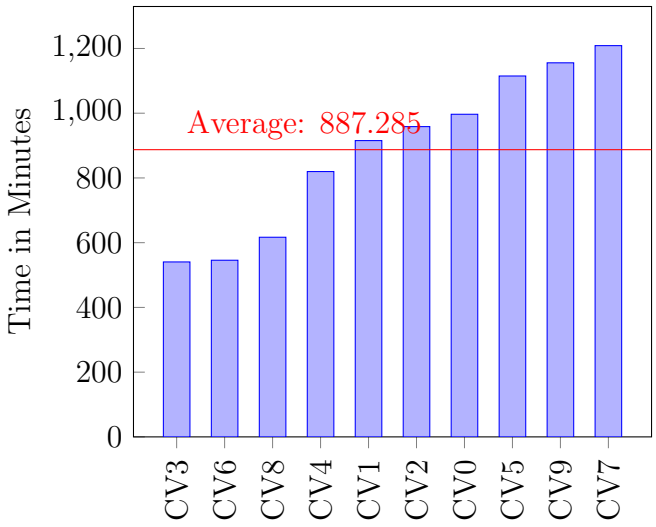

Figure 38: Time Spent Learning Using NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

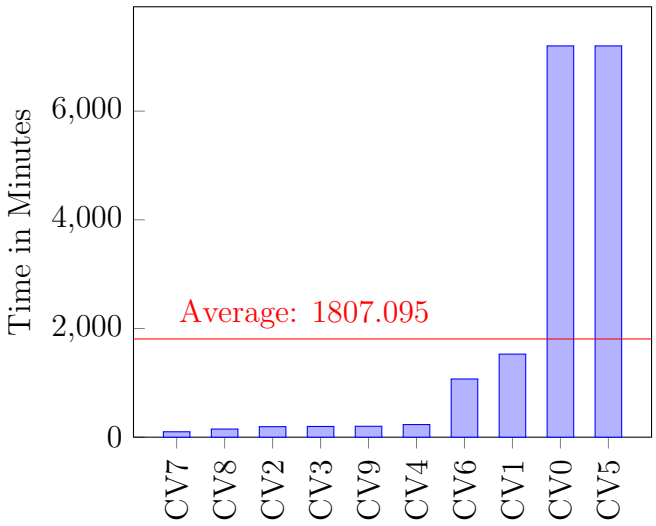

Figure 39: Time Spent Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

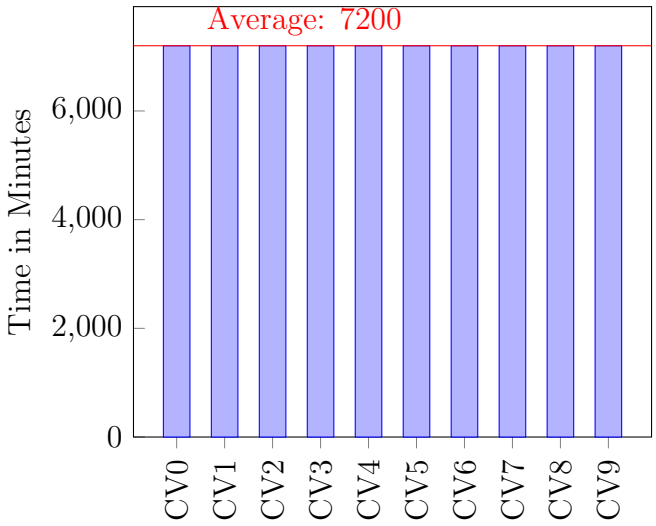

Figure 40: Time Spent Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

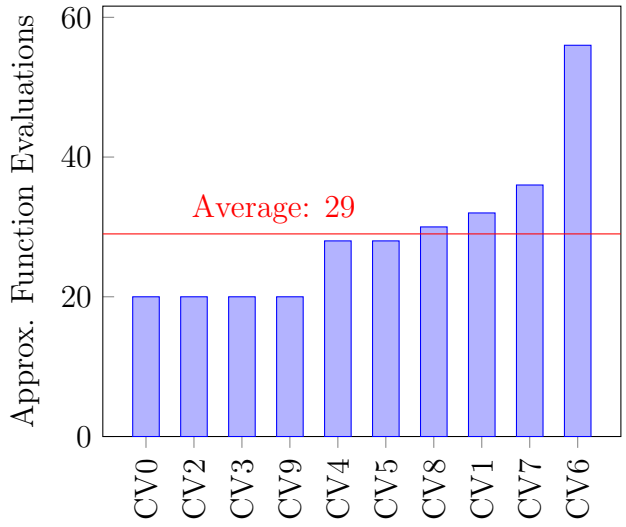

Figure 41: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

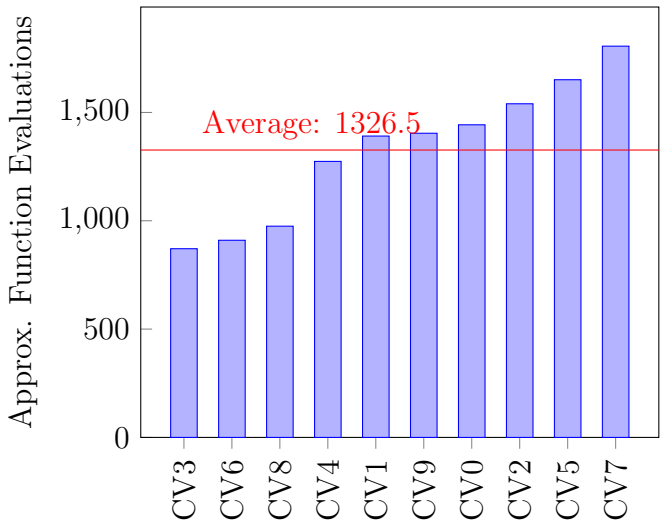

Figure 42: Approximate Number of Function Evaluations Learning Using NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

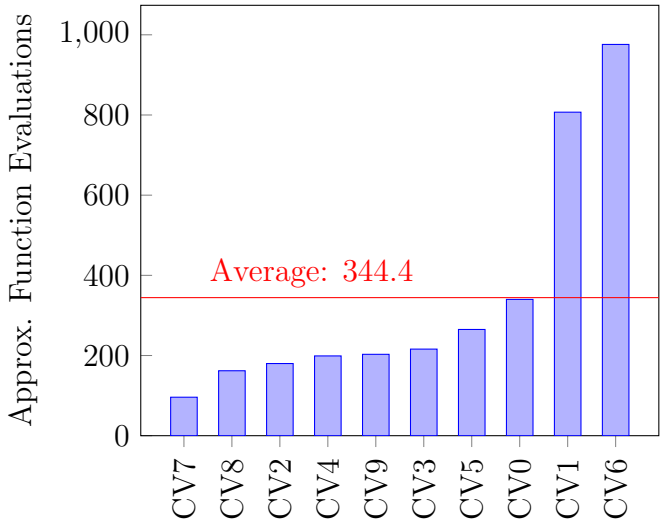

Figure 43: Approximate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

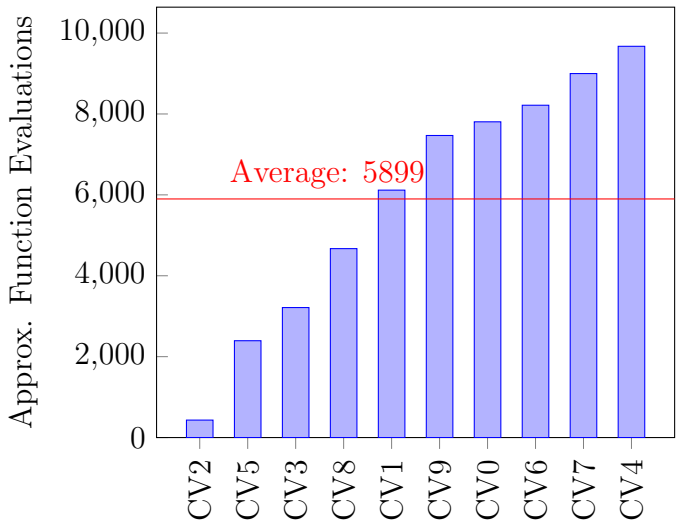

Figure 44: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on ADULT Dataset

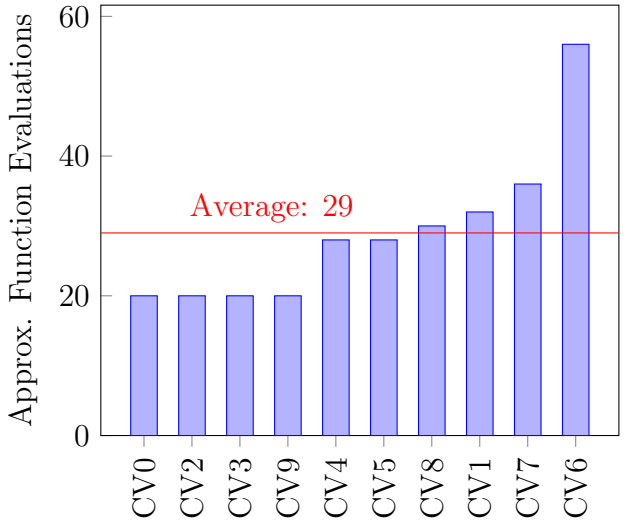

Figure 45: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

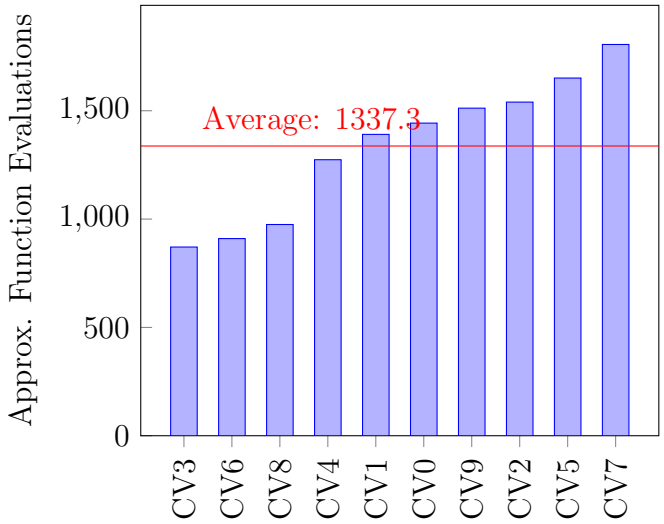

Figure 46: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

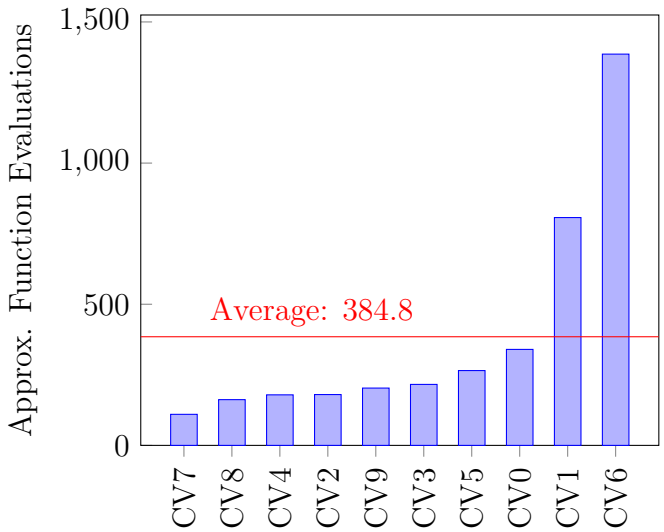

Figure 47: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

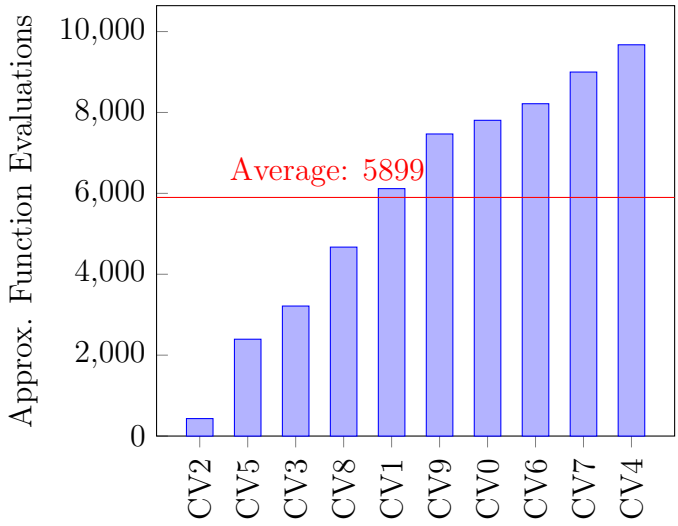

Figure 48: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on ADULT Dataset

Appendix B

Spambase Dataset Percent Correct Comparison Graphs

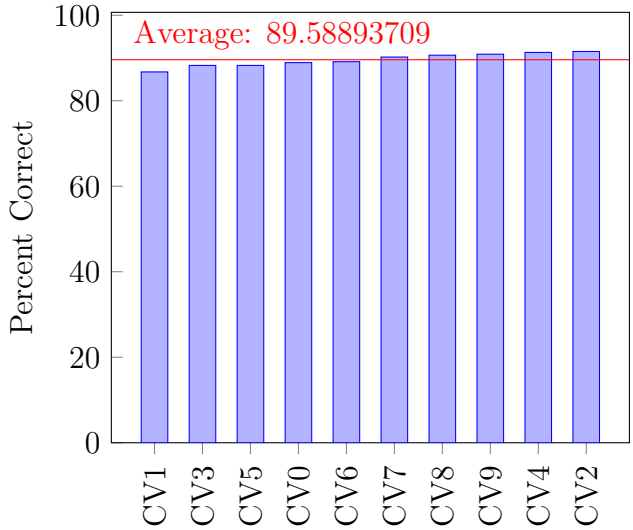

Figure 49: Percent Correct NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

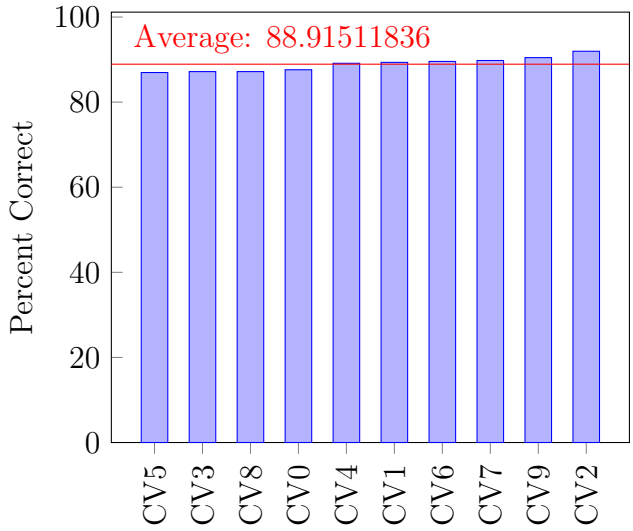

Figure 50: Percent Correct NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

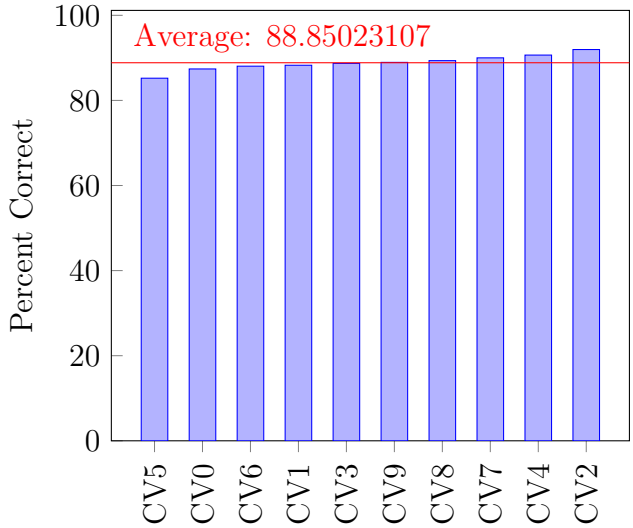

Figure 51: Percent Correct AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

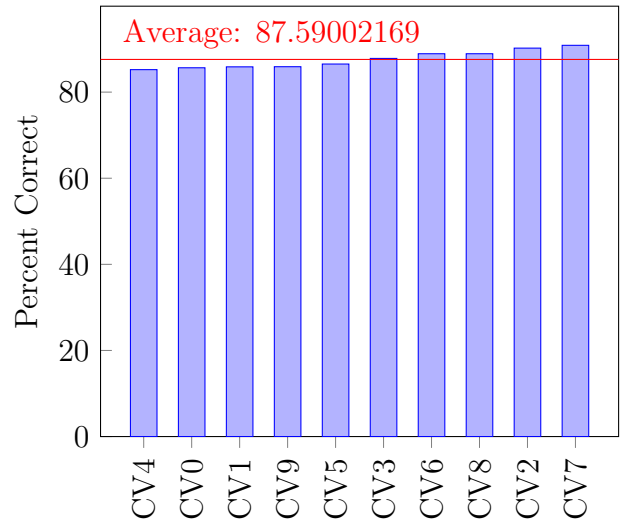

Figure 52: Percent Correct AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

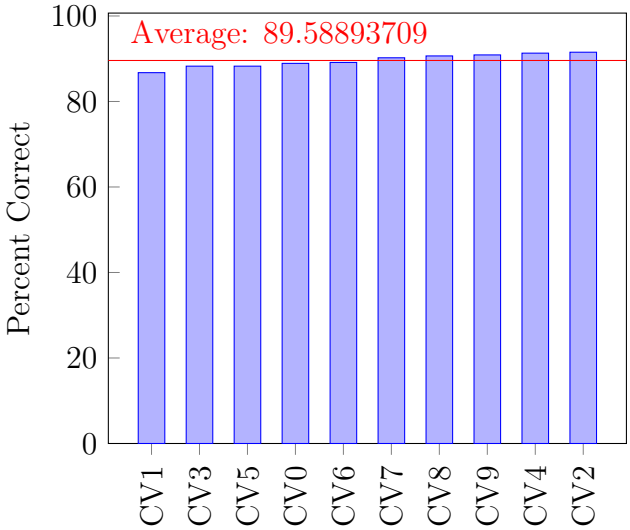

Figure 53: Percent Correct NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

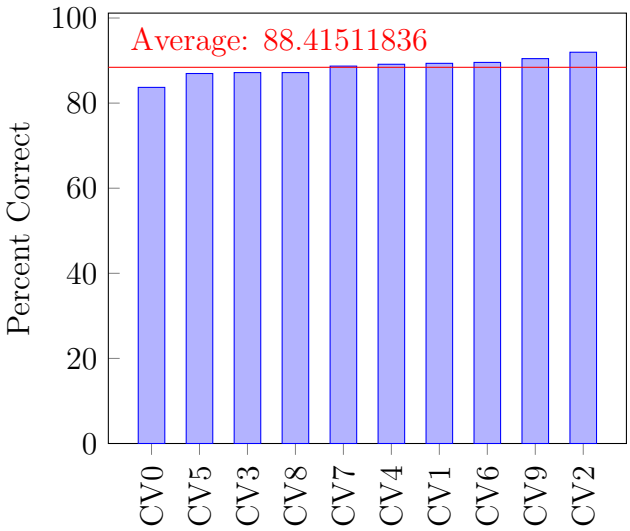

Figure 54: Percent Correct NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

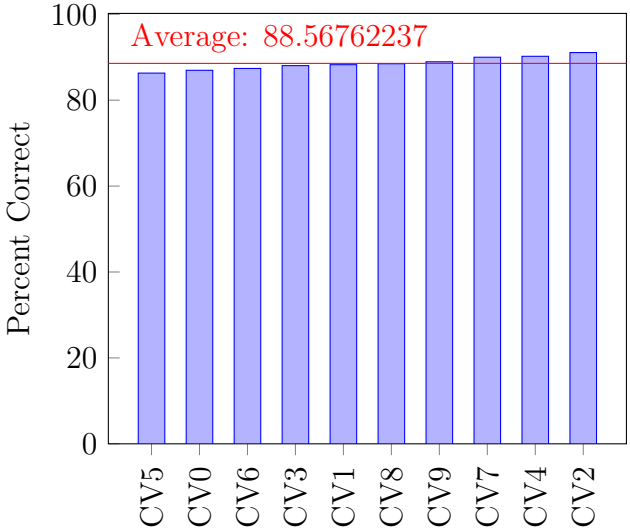

Figure 55: Percent Correct AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

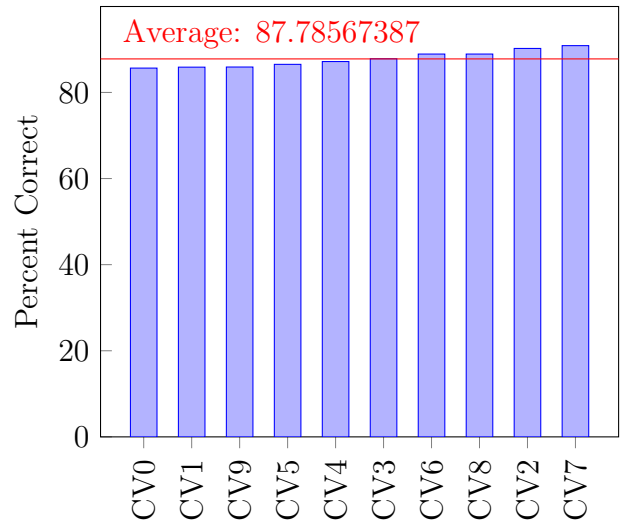

Figure 56: Percent Correct AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

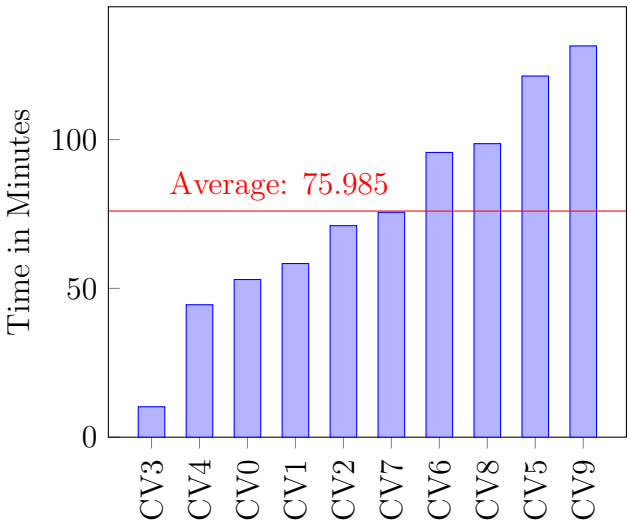

Figure 57: Time Spent Learning Using NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

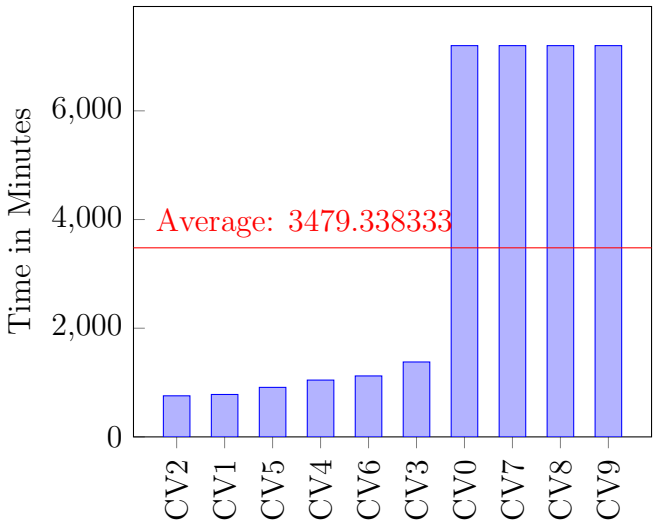

Figure 58: Time Spent Learning Using NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

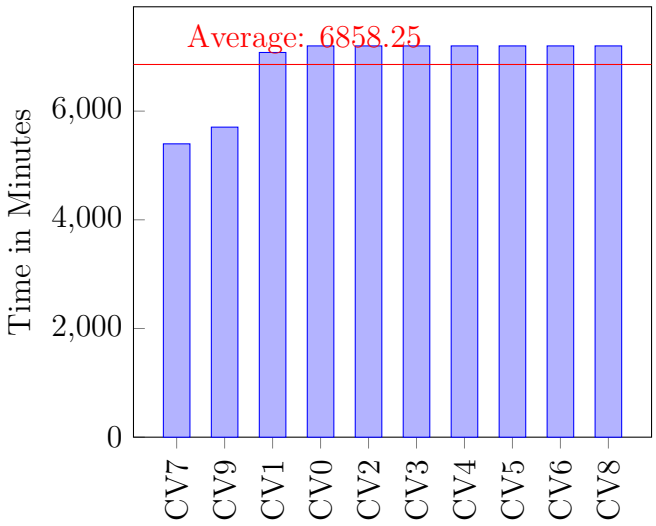

Figure 59: Time Spent Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

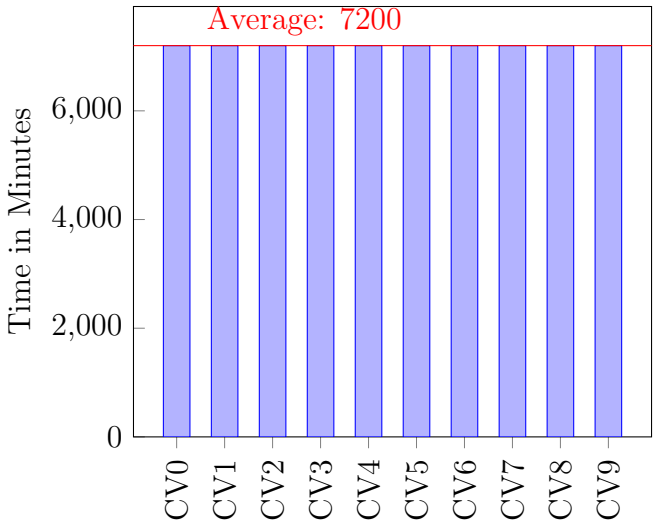

Figure 60: Time Spent Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

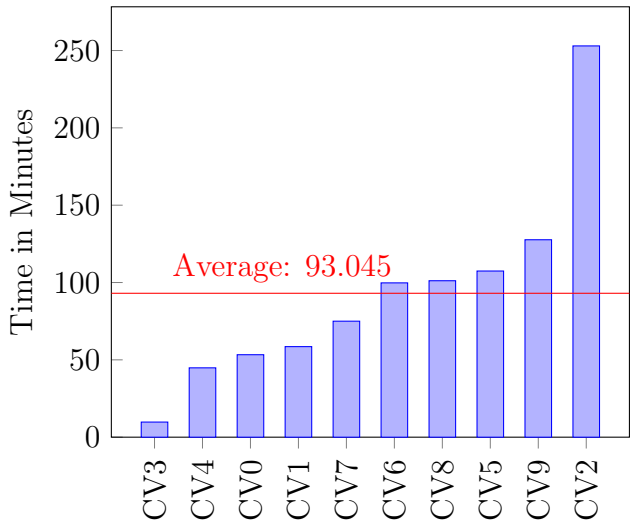

Figure 61: Time Spent Learning Using NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

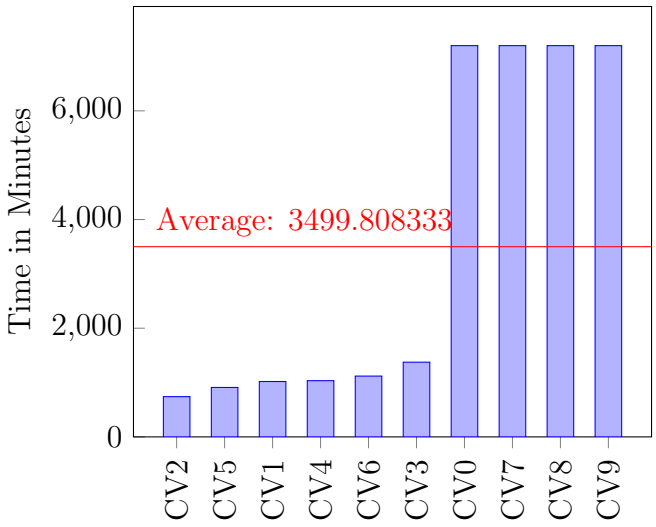

Figure 62: Time Spent Learning Using NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

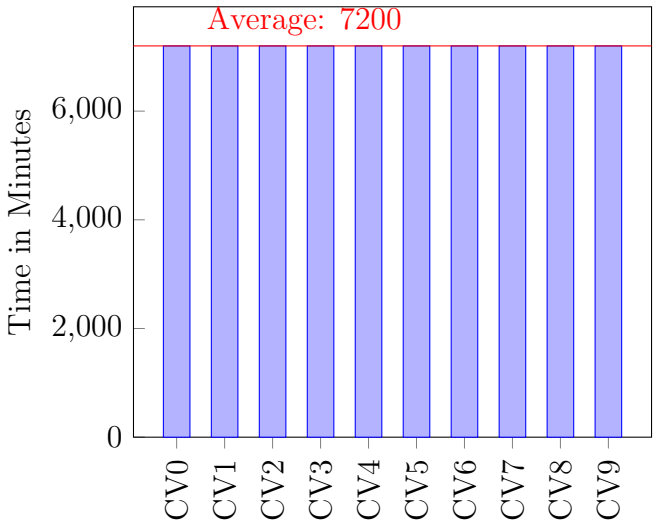

Figure 63: Time Spent Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

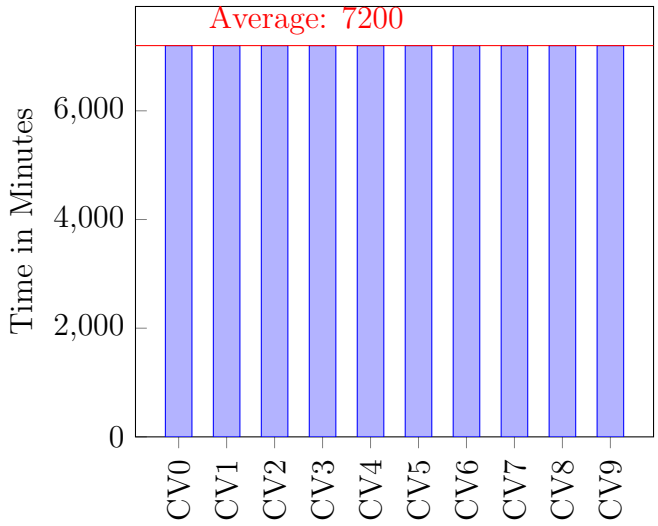

Figure 64: Time Spent Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

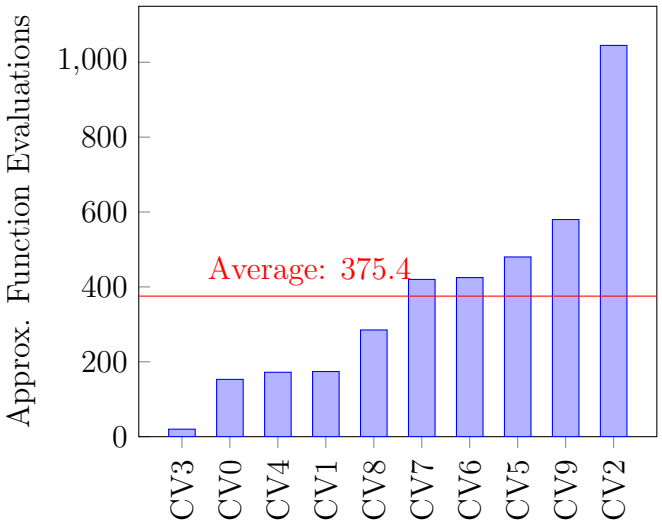

Figure 65: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

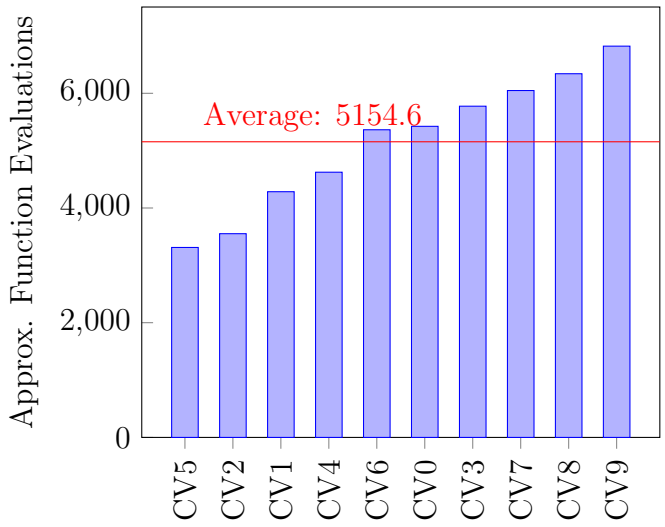

Figure 66: Approximate Number of Function Evaluations Learning Using NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

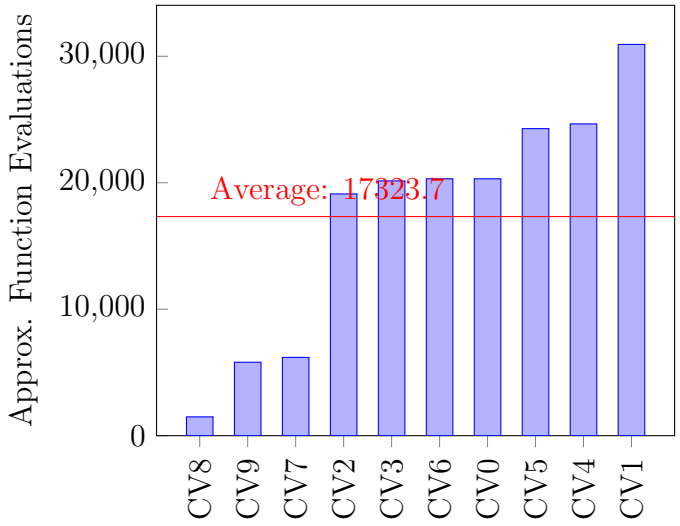

Figure 67: Approximate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

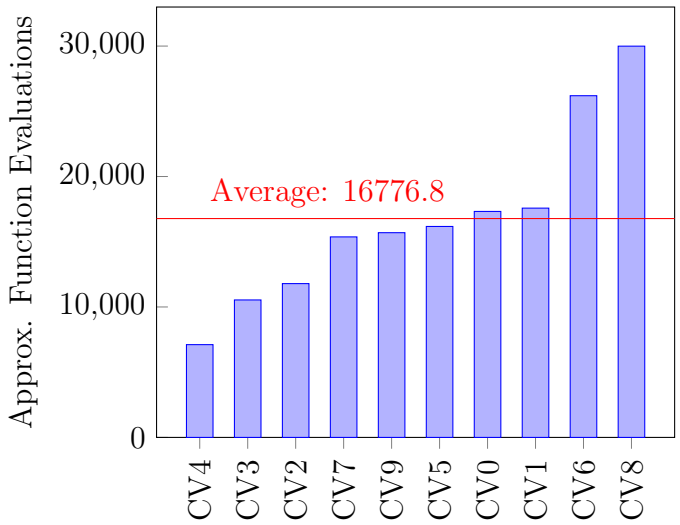

Figure 68: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on SPAM Dataset

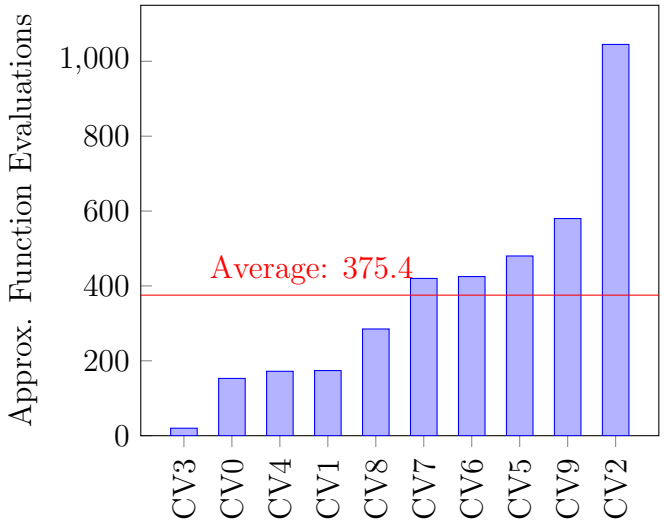

Figure 69: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

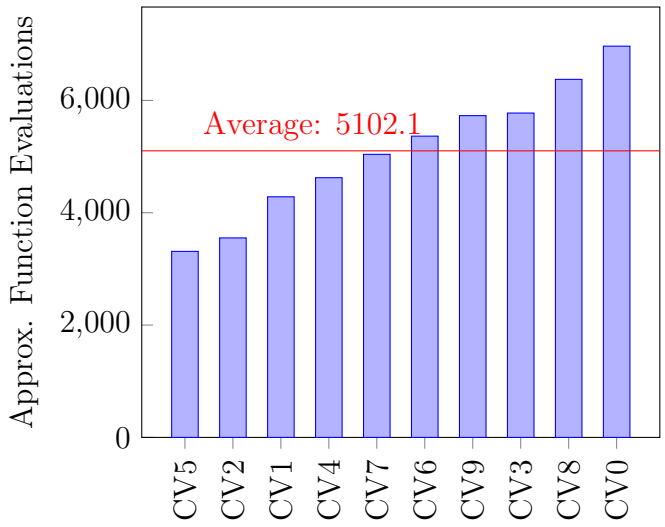

Figure 70: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

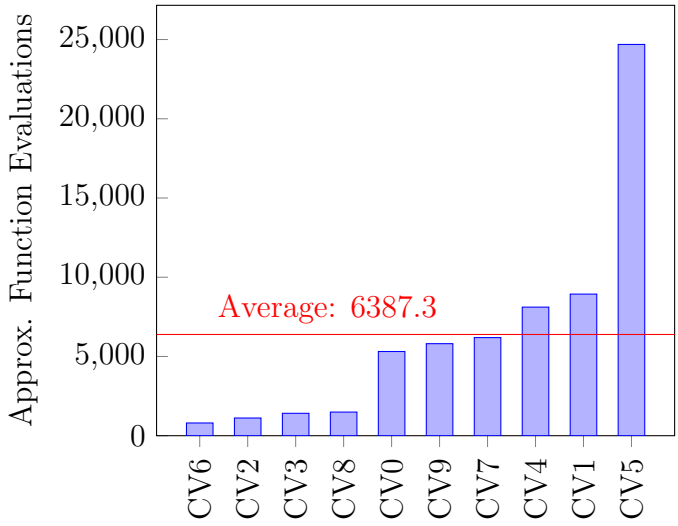

Figure 71: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

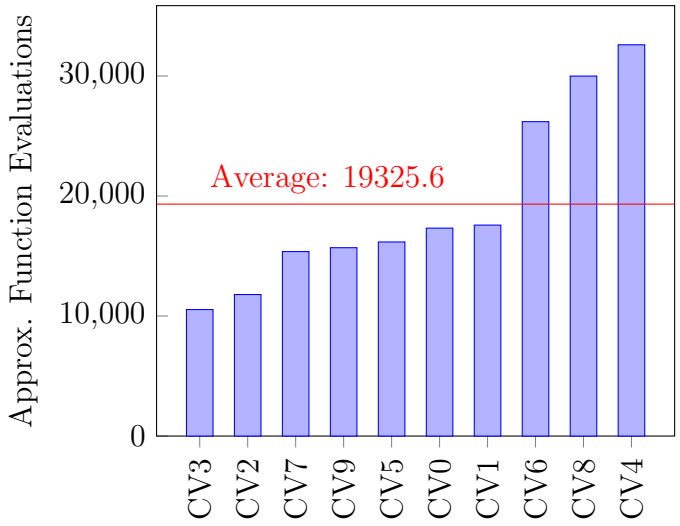

Figure 72: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on SPAM Dataset

Appendix C

Tic-Tac-Toe EndGame Dataset Percent Correct Comparison Graphs

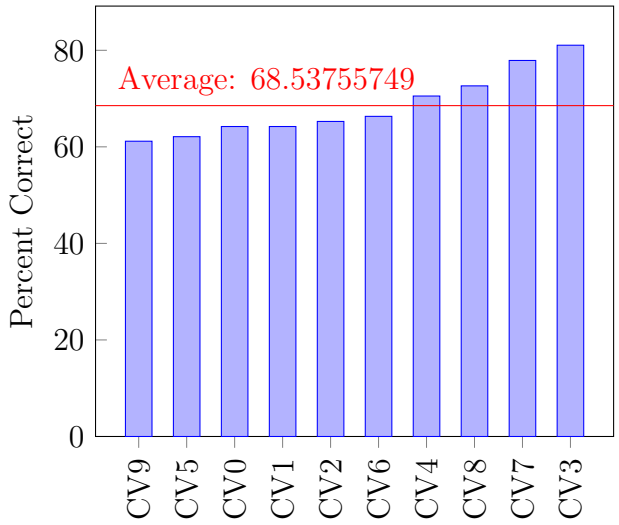

Figure 73: Percent Correct NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

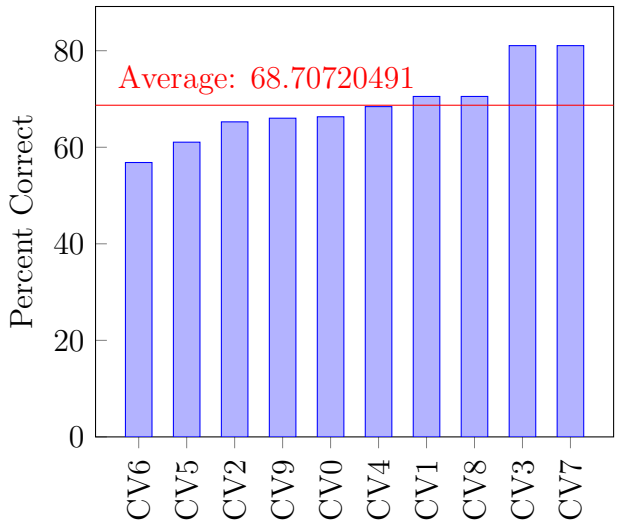

Figure 74: Percent Correct NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

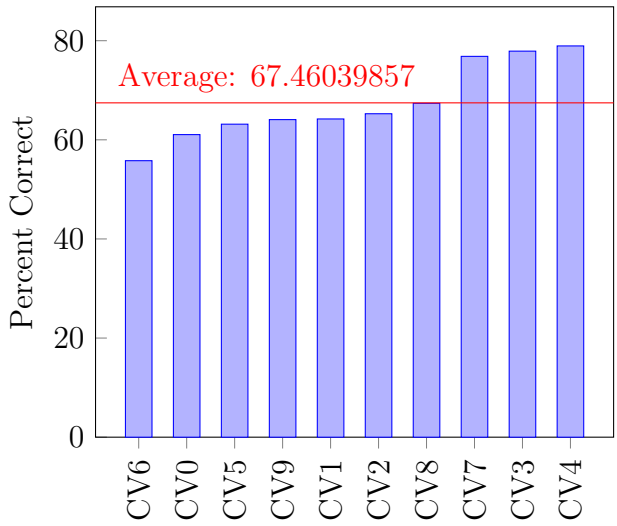

Figure 75: Percent Correct AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

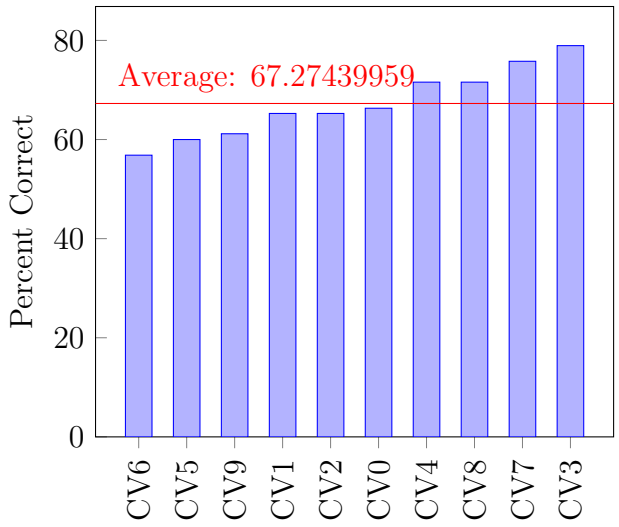

Figure 76: Percent Correct AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

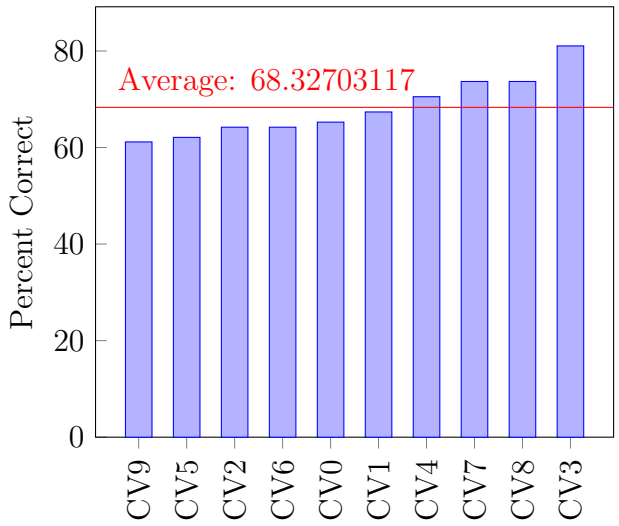

Figure 77: Percent Correct NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

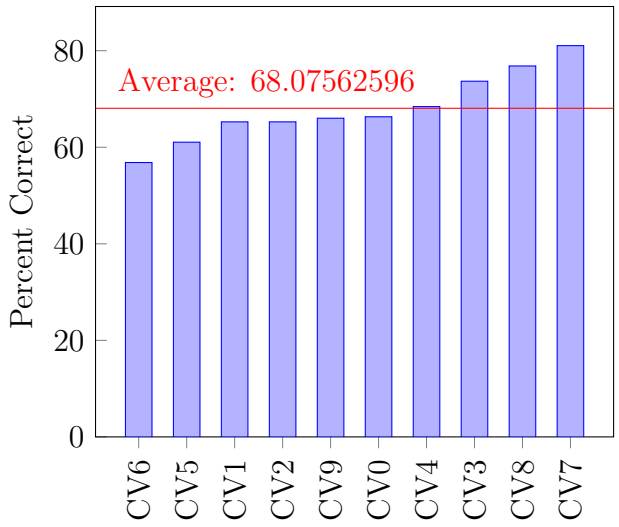

Figure 78: Percent Correct NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

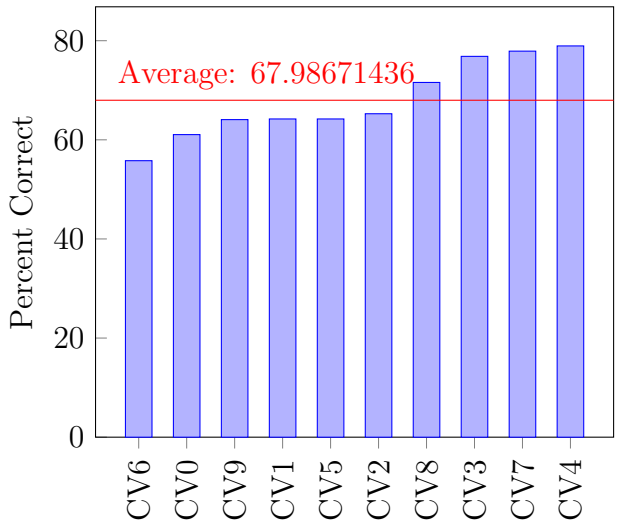

Figure 79: Percent Correct AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

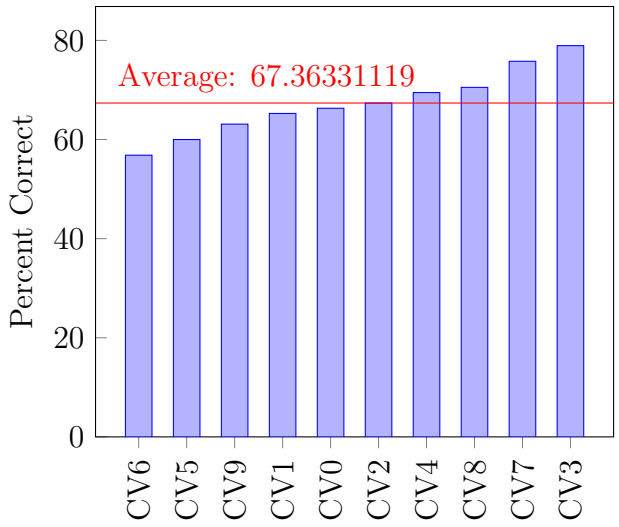

Figure 80: Percent Correct AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

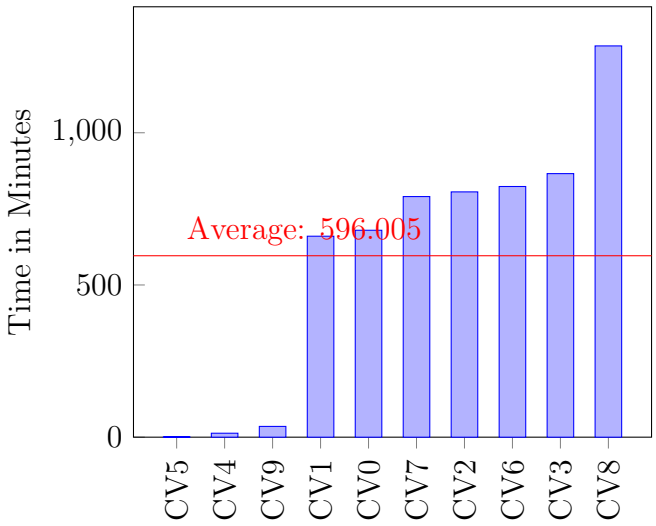

Figure 81: Time Spent Learning Using NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

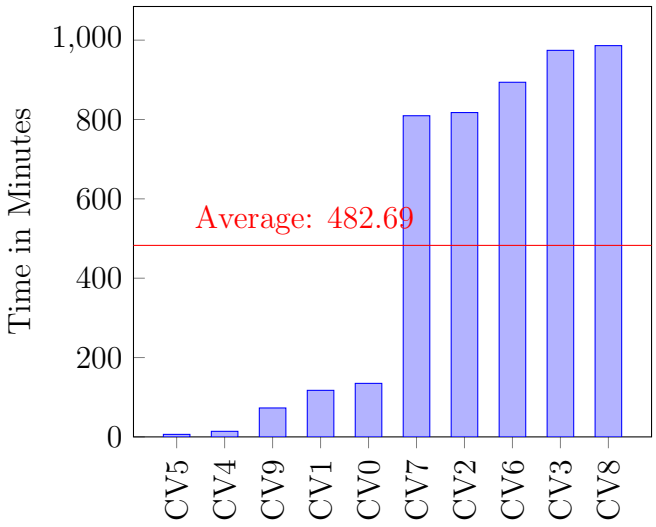

Figure 82: Time Spent Learning Using NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

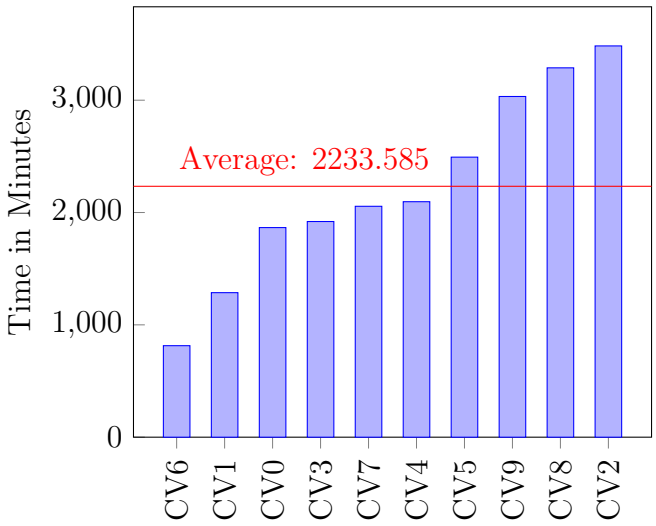

Figure 83: Time Spent Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

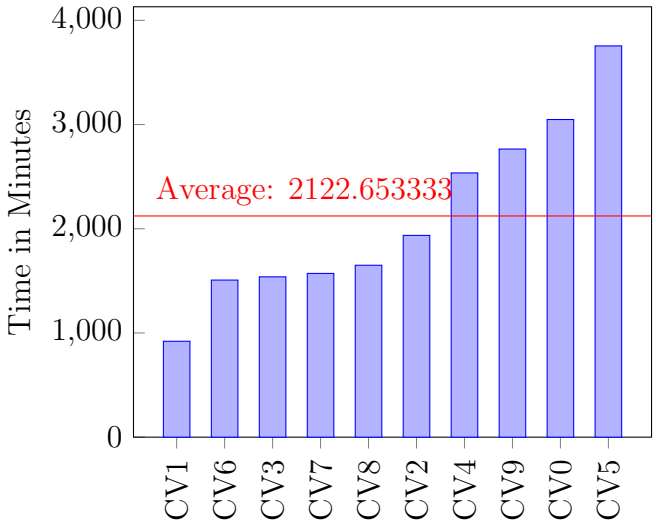

Figure 84: Time Spent Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

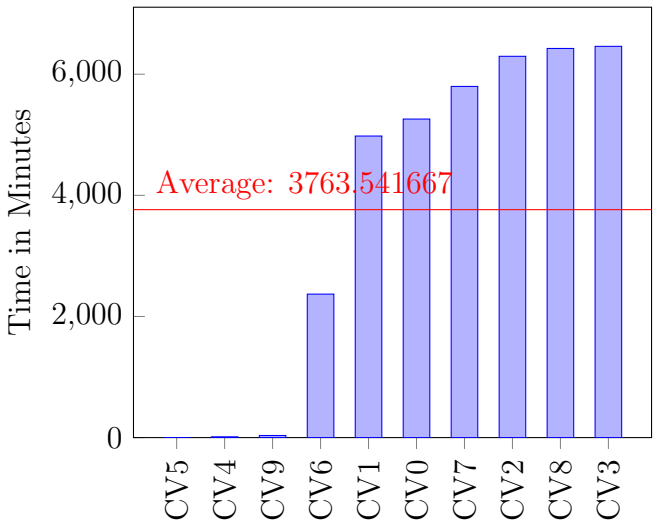

Figure 85: Time Spent Learning Using NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

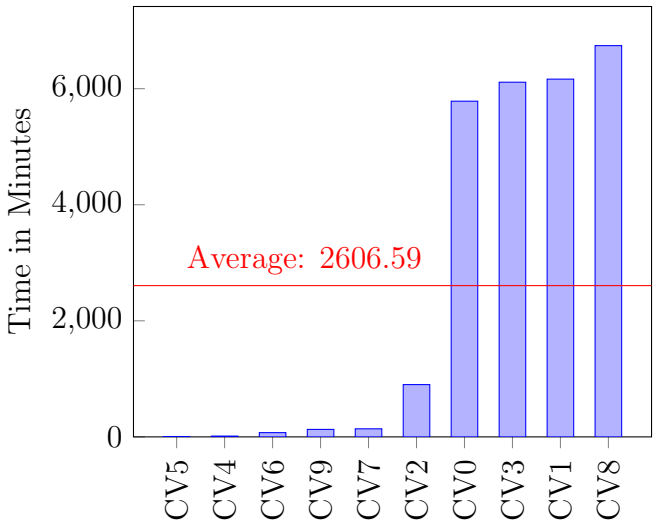

Figure 86: Time Spent Learning Using NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

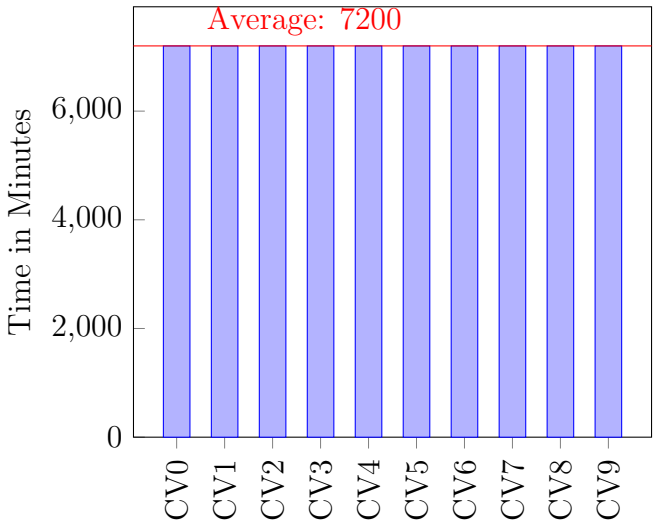

Figure 87: Time Spent Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

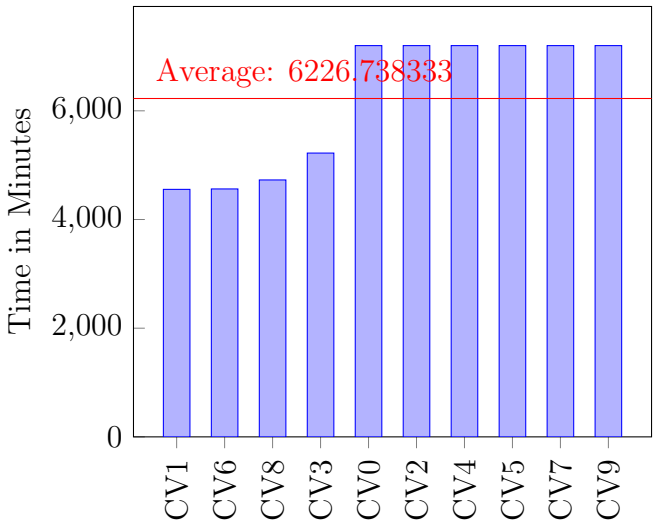

Figure 88: Time Spent Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

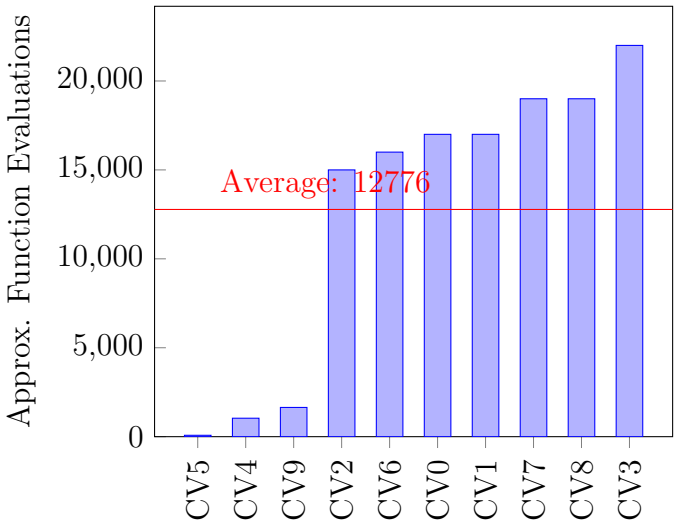

Figure 89: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

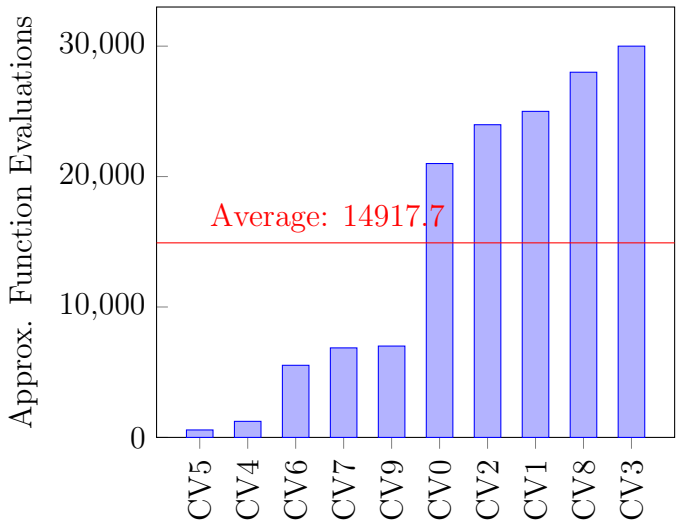

Figure 90: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

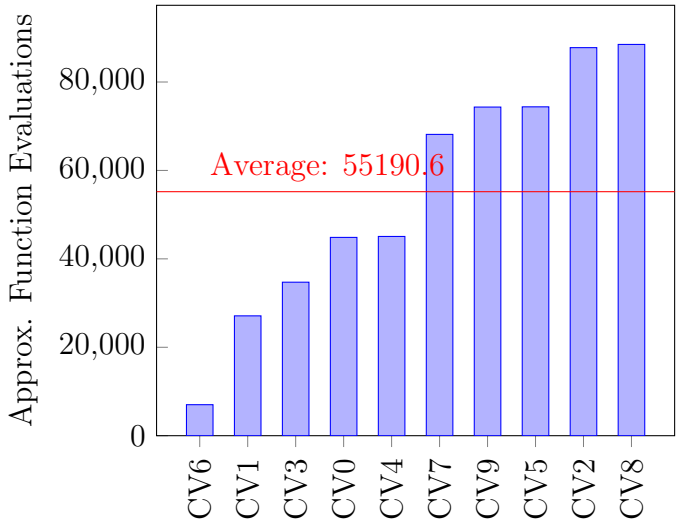

Figure 91: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

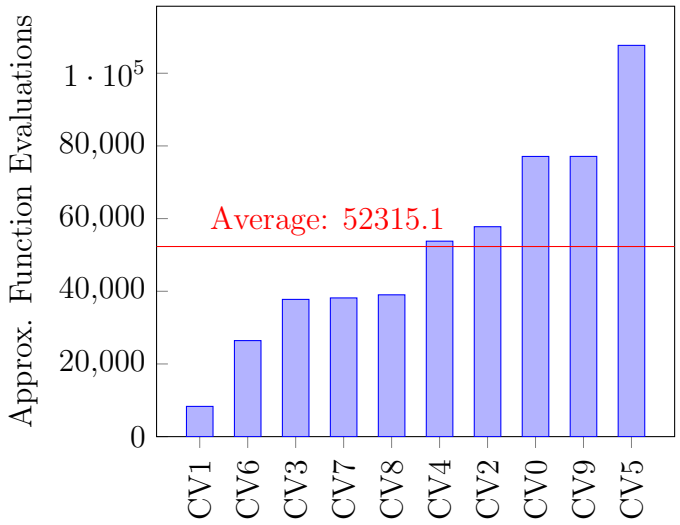

Figure 92: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on TTT Dataset

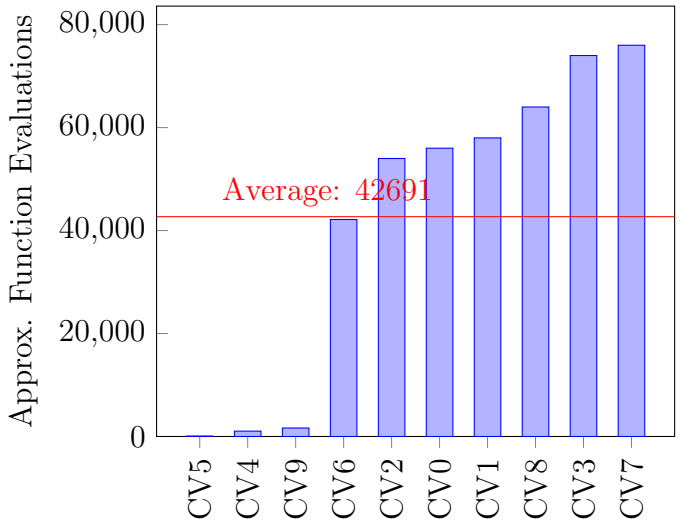

Figure 93: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

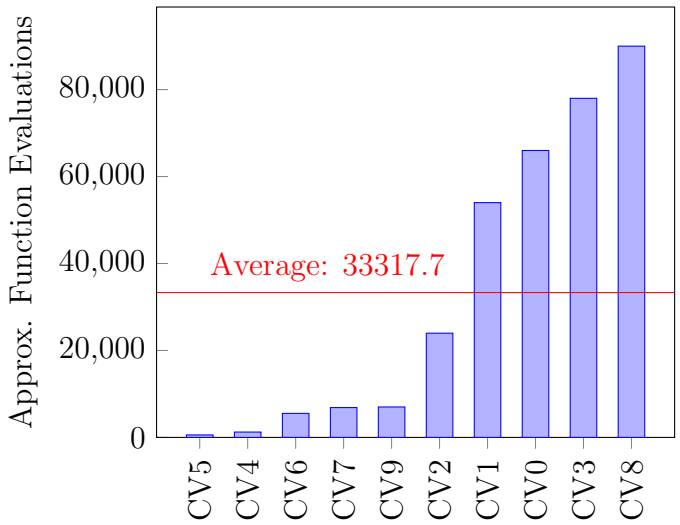

Figure 94: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

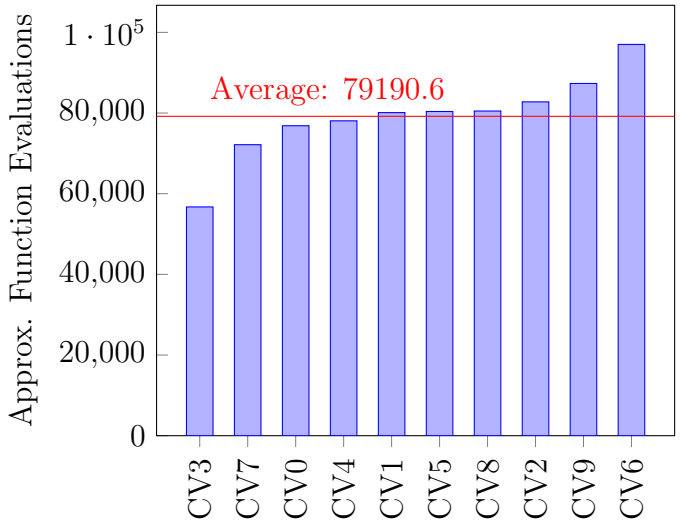

Figure 95: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

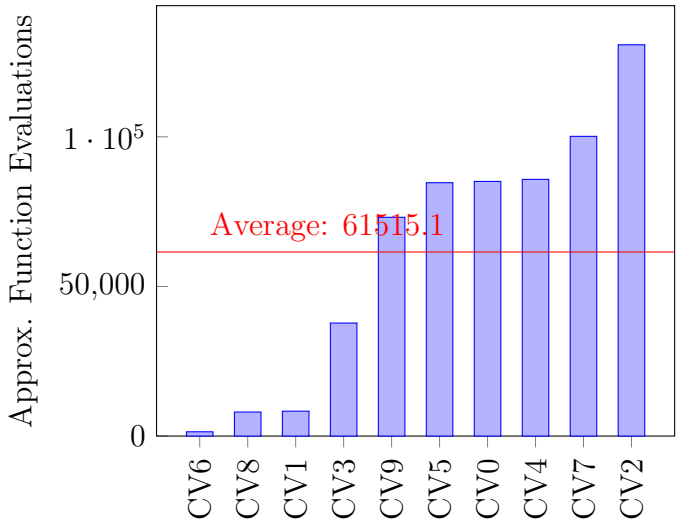

Figure 96: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on TTT Dataset

Appendix D

Breast Cancer Wisconsin (Diagnostic) Dataset Percent Correct Comparison Graphs

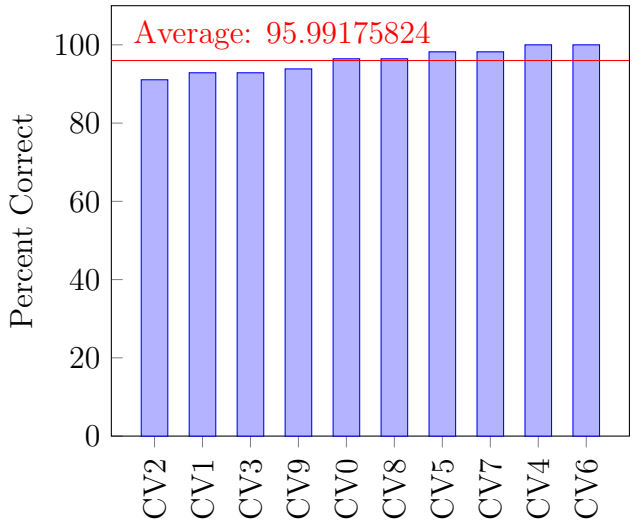

Figure 97: Percent Correct NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

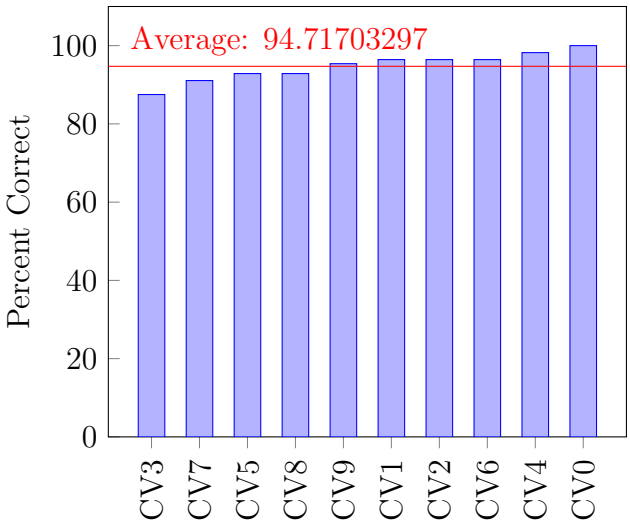

Figure 98: Percent Correct NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

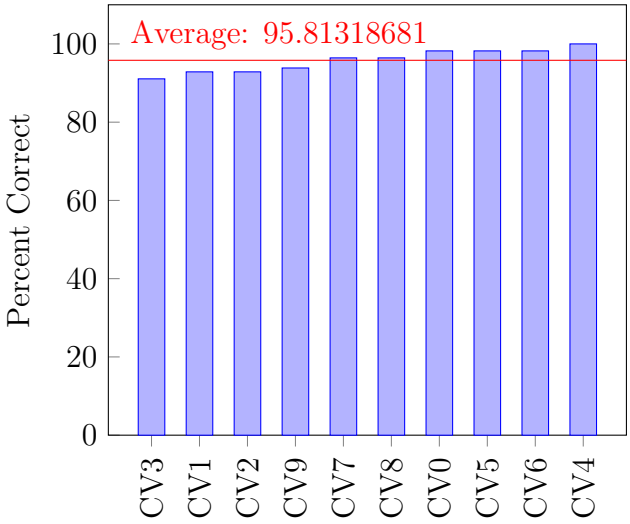

Figure 99: Percent Correct AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

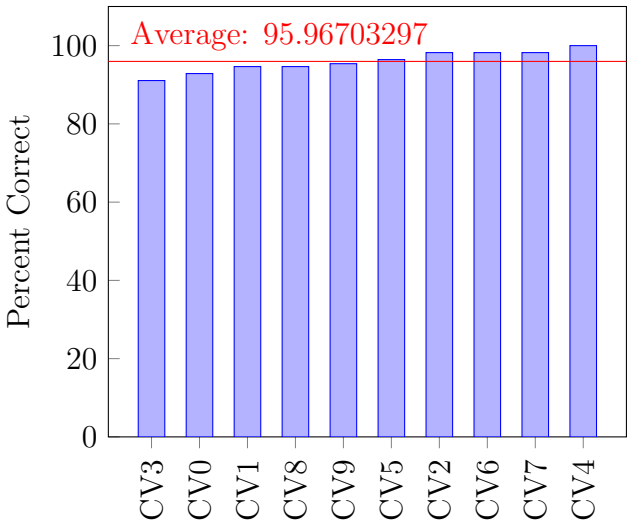

Figure 100: Percent Correct AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

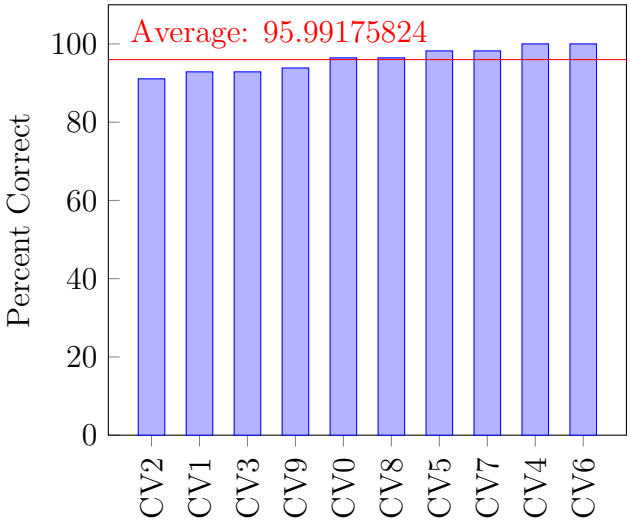

Figure 101: Percent Correct NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

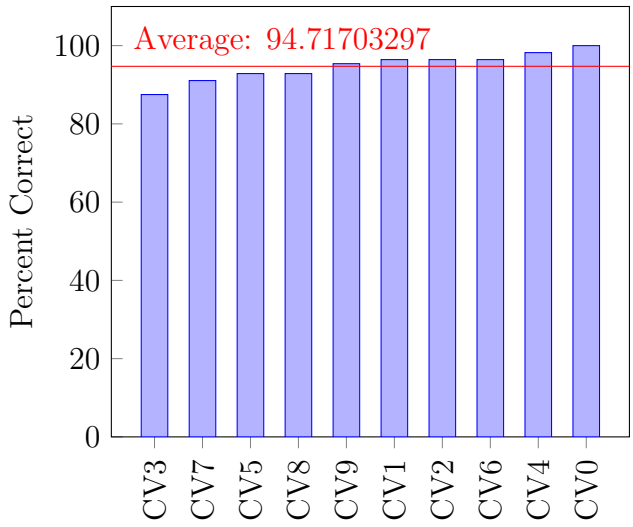

Figure 102: Percent Correct NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

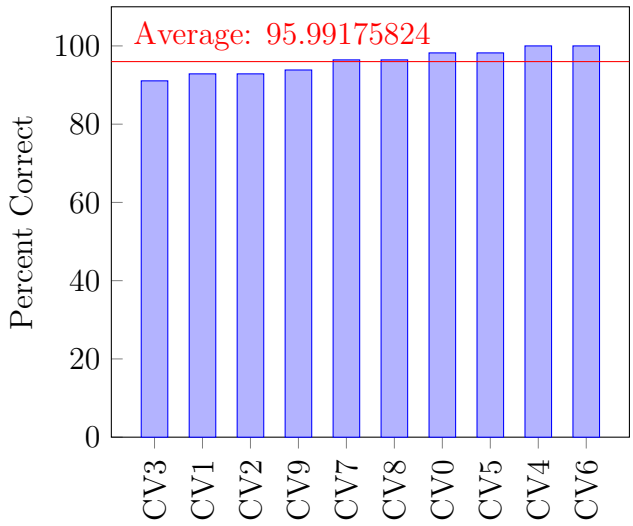

Figure 103: Percent Correct AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

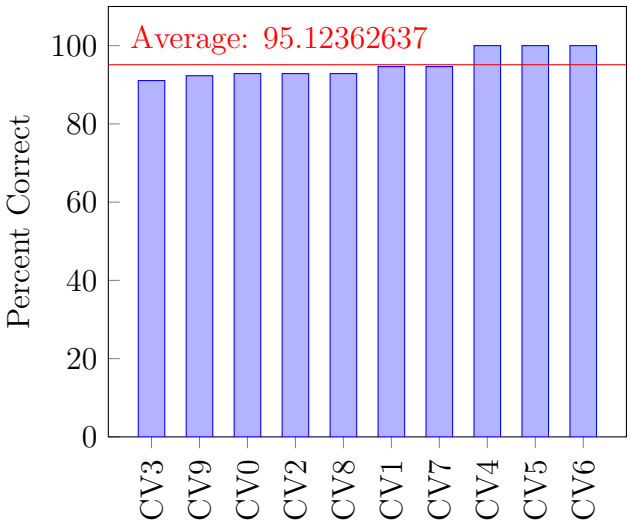

Figure 104: Percent Correct AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

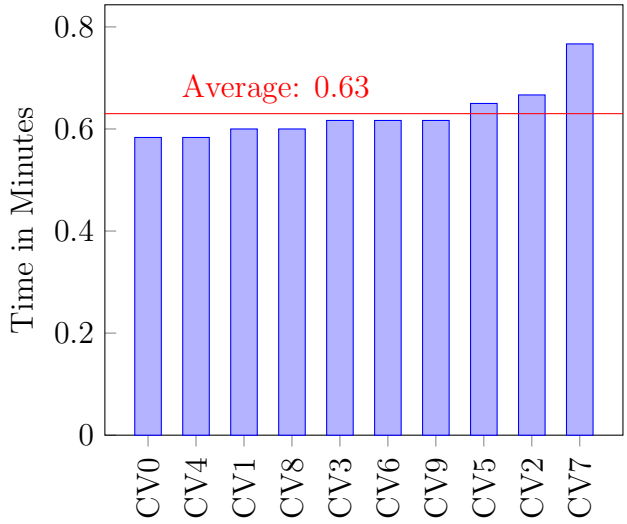

Figure 105: Time Spent Learning Using NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

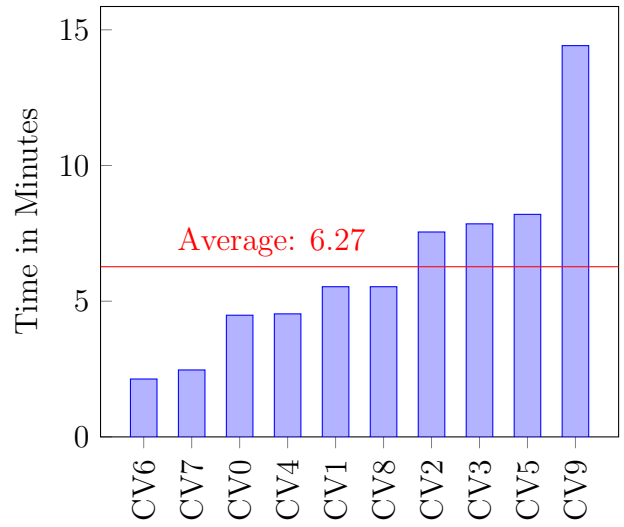

Figure 106: Time Spent Learning Using NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

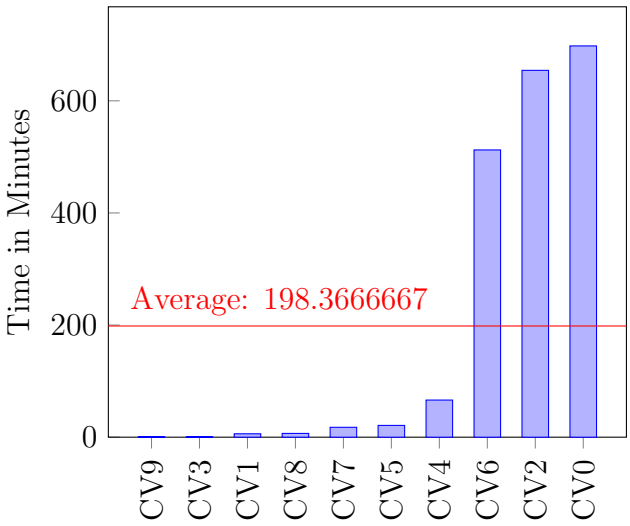

Figure 107: Time Spent Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

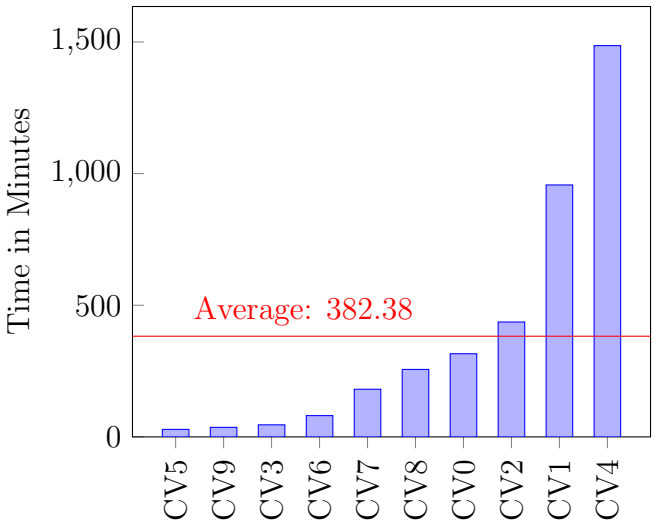

Figure 108: Time Spent Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

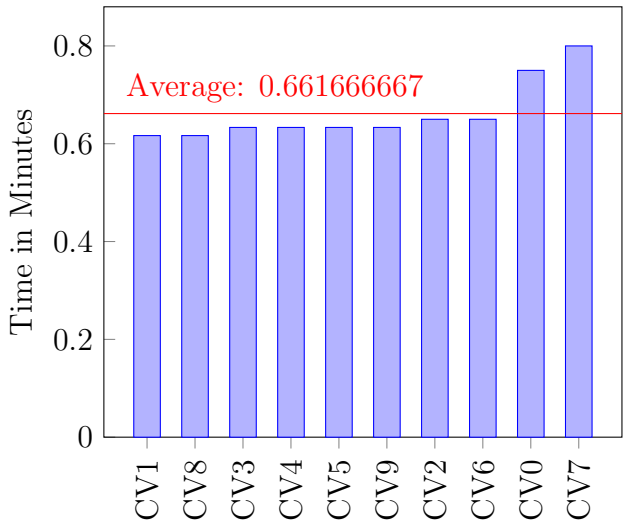

Figure 109: Time Spent Learning Using NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

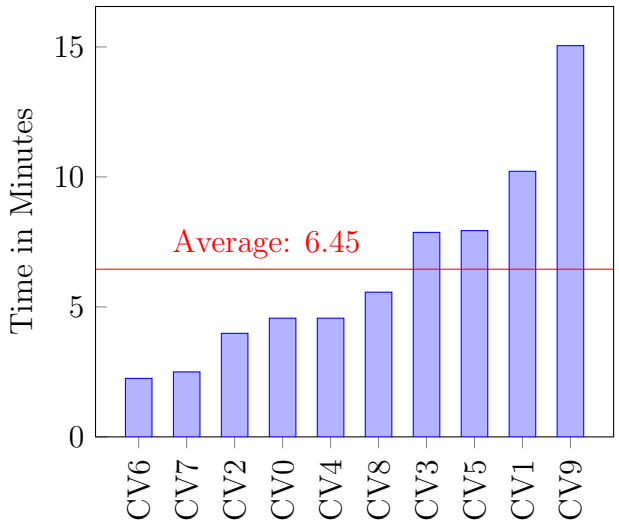

Figure 110: Time Spent Learning Using NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

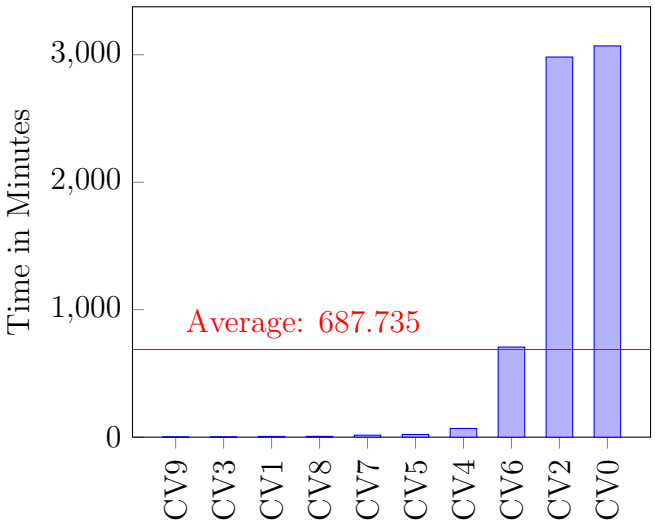

Figure 111: Time Spent Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

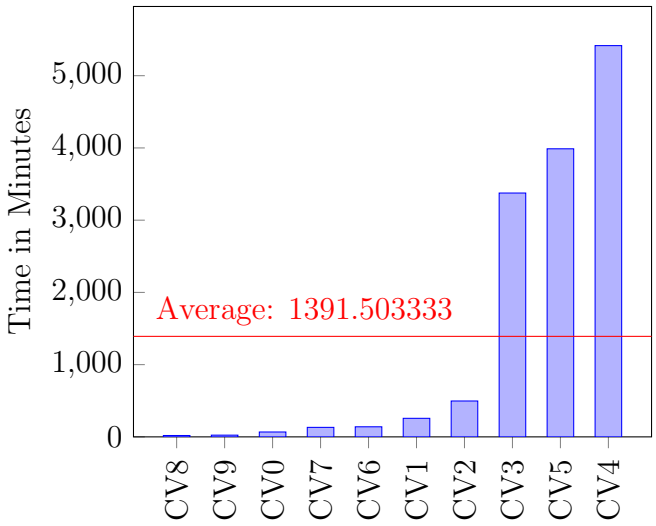

Figure 112: Time Spent Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

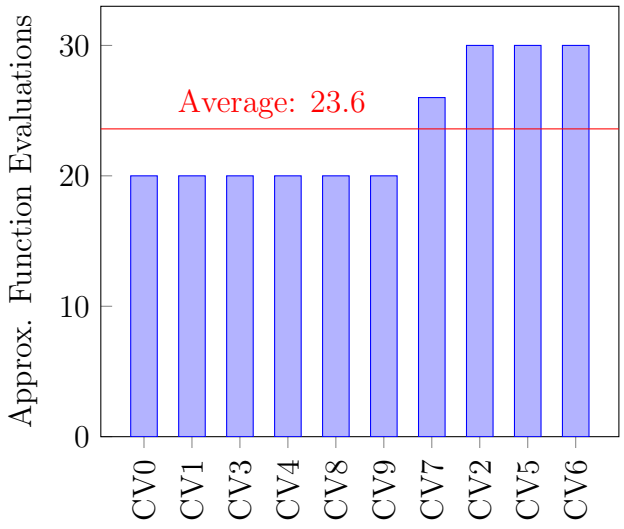

Figure 113: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

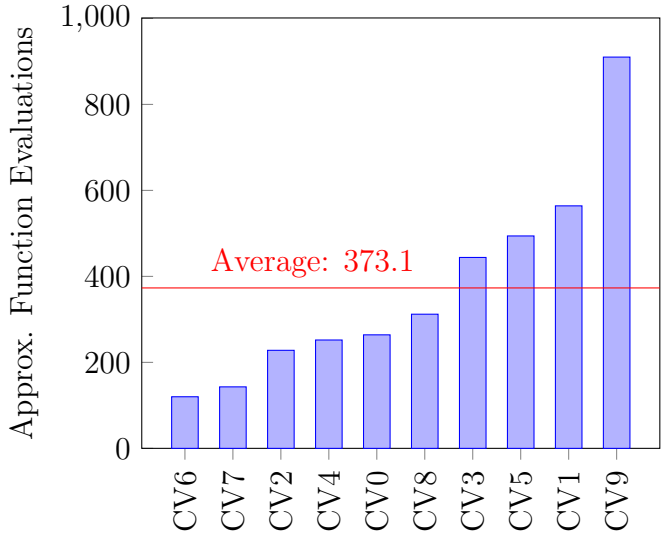

Figure 114: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

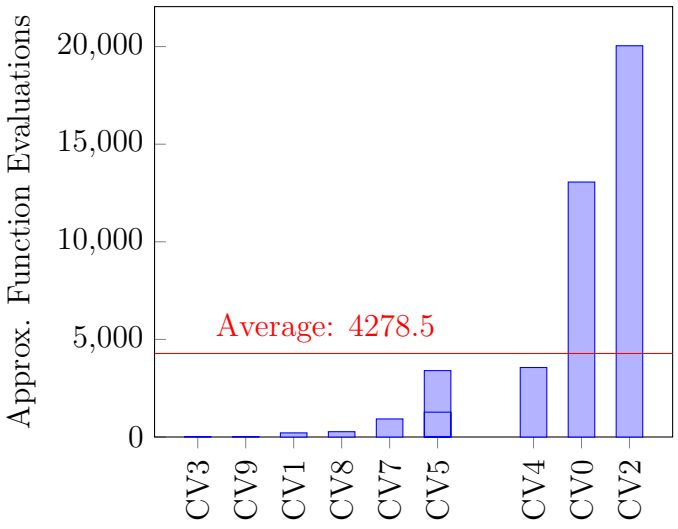

Figure 115: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

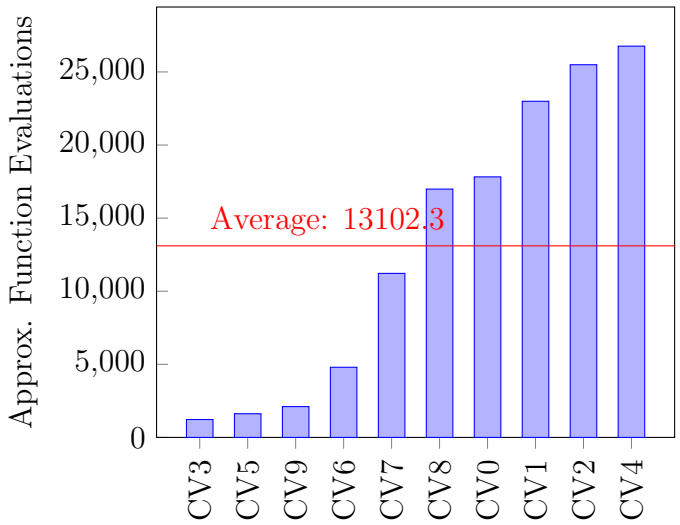

Figure 116: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 1000 Generations on Different Cross Validation Sets on WDBC Dataset

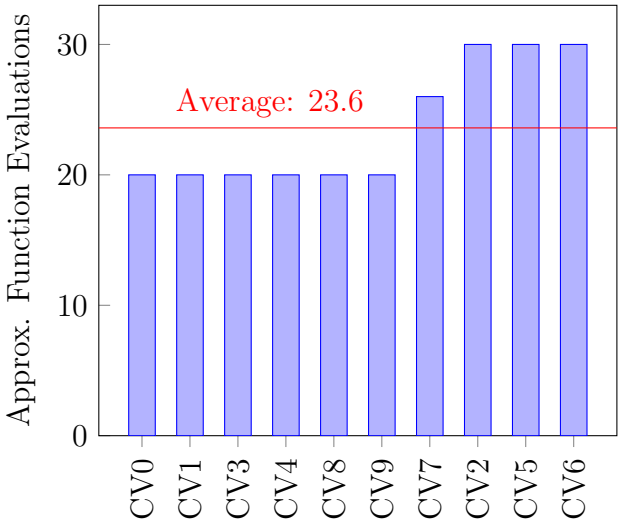

Figure 117: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

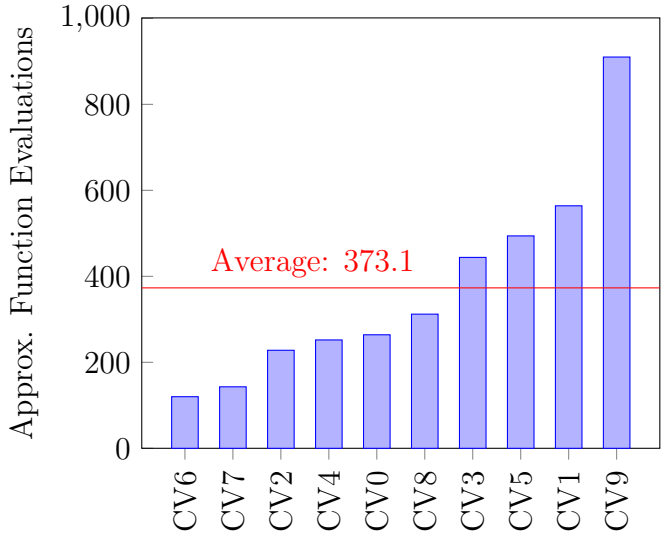

Figure 118: Approxiamate Number of Function Evaluations Learning Using NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

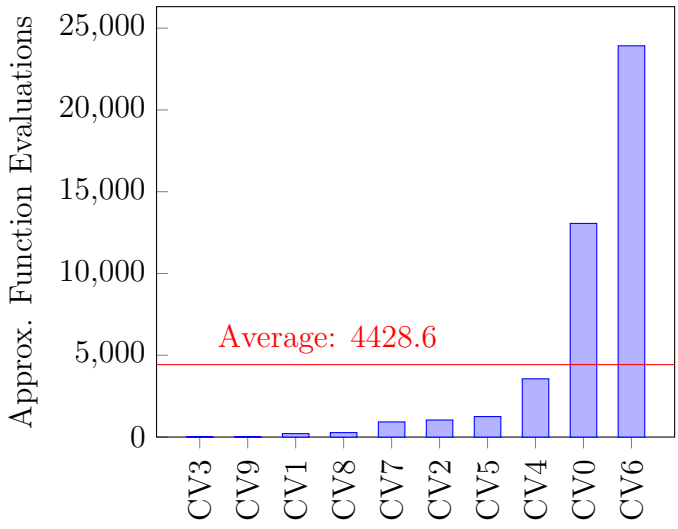

Figure 119: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 0 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

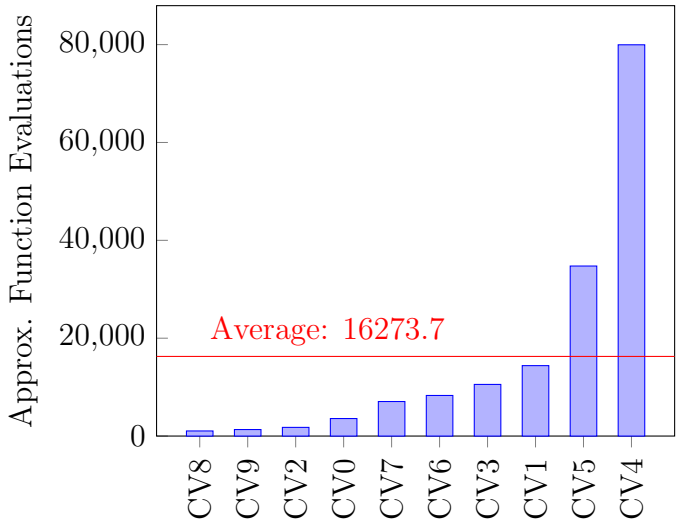

Figure 120: Approxiamate Number of Function Evaluations Learning Using AdaBoost with NEAT Starting 10 Hidden Neurons with Maximum 2000 Generations on Different Cross Validation Sets on WDBC Dataset

VITA

Robert Carl Schukei

Candidate for the Degree of

Doctor of Philosophy

Thesis: A STUDY OF ENSEMBLE LEARNING WITH ADABOOST AND NEAT

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Doctor of Philosophy in Computer Science at Oklahoma State University, Stillwater, Oklahoma in July, 2017.

Completed the requirements for the Master of Science in Applied Computer Science at Northwest Missouri State University, Maryville, Missouri in 2006.

Completed the requirements for the Bachelor of Science in Computer Science and Mathematics at Northwest Missouri State University, Maryville, Missouri in 2004.

Experience:

Instructor of Computer Science at Baker University, Baldwin City, Kansas.

Graduate Assistant for Computer Science at Oklahoma State University, Stillwater, Oklahoma.

Graduate Assistant for Computer Science at Northwest Missouri State University, Maryville, Missouri.