Extending Encog: a study on classifier ensemble techniques

Alan Mosca
MSc Computer Science Project Report
Department of Computer Science and Information Systems
Birkbeck College, University of London

September 2012

Abstract

The goal of this project is to illustrate the creation of additional components for the Encog Java Library, which implement some of the most common and widely used ensemble learning techniques. I will cover the prior knowledge required for understanding the work, including mathematical proof and discussion where possible, the design of the new components and how they fit in the overall design of the library, the procedure of validating and evaluating the performance of the newly implemented techniques, and finally the results of this evaluation.

Supervisor: Dr G. Magoulas

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ensemble learning techniques are a very powerful tool in Machine Learning. They have proven themselves time and time again to be useful for improving the performance of machine learning techniques, and can be applied to an existing process without changing the understanding of the problem too much. Thus, they have been recently used to a very large extent (the top two entries of the netflix million dollar prize were both implementations of stacked generalization - a very common and powerful ensemble technique). In parallel, there is a well-known Java library called Encog, which implements many machine learning techniques and has reached a good level of adoption. This library is, however, missing Ensemble techniques in its current implementation, and there have been several requests to add such functionality to the library.

1.1 Purpose of the project

The plan for this project was to implement the support for ensemble techniques into Encog, and to add a few of the common methodologies. As well as a design and coding exercise, it is an exploration of the various ensemble techniques available. I also planned to test the performance using some of the readily available datasets and to compare the results to some others already published. The number of machine learning techniques, ensemble techniques and test datasets is very large, so during implementation it was only realistically possible to attack a subset of the possibilities in this project. Having to choose, I picked Neural Networks as the underlying machine learning techniques because Encog was initially born as a Neural Networks package and subsequently expanded.

1.2 Background

There is a lot of literature on machine learning methods in general and on neural networks in particular, and on ensemble learning methods. The relevant chapters will cover the prior work more in detail.

1.2.1 Machine learning problems

Machine learning techniques are varied and can be applied to many different types of problems. These problems are normally categorised by lerning method and by output type.

Supervised learning

Supervised learning is a method of learning where a *training* set is provided along with the expected output for each element of the training set, such that the learning algorithm can generate a system that will perform predictions on new sets presented at a later date. A *test set* is also provided with similar characteristics (but with independent data from the training set), so that it is possible to compare the predicted outputs with the correct ones, and approximate the level of error of a certain algorhithm. Another sub-classification is made based on the type of prediction:

- *Classification problems.* With classification problems, we are presented with a number of input attributes and we are expected to predict which "class" these inputs belong to. This is a powerful technique for pattern recognition and decision making.

- *Regression problems.* Regression problems still make predictions, but rather than predicting that a set belongs to a certain class, they predict an unbounded output value. This is a poweful technique for predicting open-ended data, like prices or rainfall (and many others).

Many of the supervised learning techniques can be used both for classification and regression, with only very small adaptations.

Unsupervised learning

Unsupervised learning techniques are generally presented with a set of data that do not have any associated prior results (this is often referred to as unlabeled data). The machine learning algorithm is expected to produce output (e.g. classification) solely based on patters from the input data.

- *Clustering.* Clustering is an unsupervised technique used to extract class values from the way that the data tends to cluster around certain values. There are several different techniques, but most of them tend to make at least some assumption about the output that is expected (e.g. the number of classes).

- *Generative models.* Generative models are used in machine learning to simulate the distribution of a variable. These technique will "learn" a distribution and produce (generate) new samples according to this internal model.

- *Dimensionality reduction.* Dimensionality reduction can be defined as a method that reduces the number of features in a specific data set, with the intent of improving performance and accuracy of a machine learning method. This can be done either by feature selection, which tries to discard some features that may not be very significant (or indeed just noise),

and feature extraction, which tries to combine the information contained in two or more features, into a smaller number of features without losing data.

1.2.2 Online vs Offline learning

We can decide to train our machine learning systems once before using them (offline learning), or we can feed back the information from inputs that arrive while predictions are being made (online learning). This second technique requires some way of providing feedback to the system, and the training methods need to be adapted so that the system is updated quickly enough so as to not affect the performance by too much.

1.2.3 Ensemble Learning

Ensemble learning combines many different learning techniques in order to improve the proficiency of predictions, and is the main focus of this paper. If applied correctly it is a very powerful approach and can improve machine learning systems that were previously considered unusable to the point that they are widely adopted. There are several techniques and they will all be covered in their own chapter.

1.3 Approach

The project can be divided in different focus areas:

1. The first part of the project focused on implementing the support for ensemble techniques in Encog, trying to alter the original structure of the library as little as possible. This will enable other contributors to add new ensemble techniques easily following an agreeable standard.
2. To implement some of the most common techniques and methods on top of this basic framework design, with the dual purpose of testing the framework and providing working examples.
3. The creation of a benchmarking environment, including the support for running long jobs. This is an essential part of the implementation, because it will provide immediate feedback on the performance of the library
4. Investigation of some datasets for benchmarking. We require some well-known data to assess correctness of the techniques and also provide a comparison between each approach.
5. Benchmarking is the last part of the project, and its success depends on all the other tasks being completed correctly and efficiently.

1.4 Structure of this document

1.4.1 Review

In the first two chapters I explore more in detail the existing knowledge and literatrue, and present some of the most common approaches. Chapter two

explores Neural Networks and classification problems and gives an overview of the current techniques. The third chapter gives an overview of both the history of ensemble learning and the current most known approaches. The fourth chapter explains some of the internal architecture of the Encog library and looks for potential ways of implementing new features in a low-disruption fashion.

1.4.2 Implementation

Chapter 5 will cover the changes to the library, with an explanation of the architecture of the new components, and how they interact with the pre-existing code.

1.4.3 Benchmarking

The first part of chapter 6 is about the architecture of the benchmark system, and all the support infrastructure for running benchmark jobs. The second part focuses on selecting the right datasets and parameters, and providing a comparison for how single Neural Network perform on these datasets. Chapter 7 presents the results for all the various benchmarks on each of the implemented techniques, with a commentary on the observed features of these results.

1.4.4 Conclusions

Chapter 8 looks at the evidence presented in the previous chapters and draws conclusions on these ensemble techniques and their implementation, and presents some of the possibilities for expansion and revision of this work.

Chapter 2

Using Artificial Neural Networks as classifiers

This chapter covers the basics of the underlying techniques of neural networks. For a deeper discussion of the subject, please refer to the vast literature on the subject, for example [3, 38]. Neural networks are just one of many machine learning algorithms that implement classifiers. Over time many different types of neural networks have been developed and evolved, with varied architecture, function, scope and performance.

Notation In this chapter I use some notation conventions, which are explained along in the chapter. I include a quick reference table here for convenience:

- X: a vector of inputs
- W: the weights vector characteristic of a neuron
- n: the number of input values, or the dimensionality of X and W
- $f(X)$: the function implemented by a neural network
- y: the output of a neuron
- t: the weighted sum of the inputs of a neuron
- $A(t)$: the activation function of a neuron
- $P(t)$: the logistic function $P(t) = \frac{1}{1+e^{-t}}$
- Y: a vector of outputs
- z: the number of output values, or the dimensionality of Y
- $C(Y)$: the classification function
- $c(y)$: the output rounding function for classification
- Y': the vector of classification decisions
- $f_c(X)$: the function implemented by a classifier
- m: the number of instances in a data set

2.1 The basic neuron

Virtually all neural networks are based on an elementary component, the neuron. The simple artificial neuron is modelled around the behaviour of the biological neuron, found in the brains of humans (and animals). The neuron has few fundamental characteristics:

- A vector of n inputs $X \in \mathbb{R}^n_{0..1}$
- A vector of numerical weights $W \in \mathbb{R}^n$, each associated with an input
- An output $y \in \mathbb{R}_{0..1}$

If we call x_i the i^{th} input, w_i the associated weight for that input and n the number of inputs of the neuron, we can say that most neurons will implement the function

$$y = A(\sum_{i=1}^{n} w_i x_i)$$

[1] where $A(t) \in \mathbb{R} \to \mathbb{R}_{0..1}$ is the activation function of the neural network. A very common activation function is the sigmoid activation function:

$$P(t) = \frac{1}{1 + e^{-t}}$$

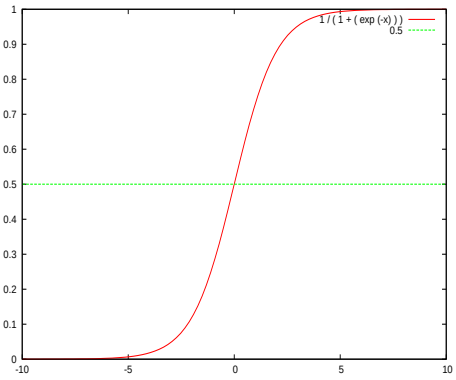


Figure 2.1: The sigmoid activation function

Applying this function to the sum of the weighted inputs puts the output in the correct range (0..1), allowing us to use the output of one neuron as the input to another. This is the basic building element for constructing neural networks. [2]

[1] We could vectorize this with a weight vector W and an input vector X and obtain the simpler notation $P(W\dot{X}')$. This is common practice in Machine Learning notation, both for simplicity reasons and because many implementations are vectorized to optimize for speed.[37]

[2] There are some variants of Neurons that use the range [-1 .. 1] instead of [0 .. 1] in both the inputs and the output. This can be considered to be equivalent from a functional perspective and is mostly just a matter of preference.

2.2 Networks

The fact that the output of a Neuron is designed to be in the same range of its inputs, makes it easy to build networks of many neurons with very diverse topologies and characteristics:

- The multilayer perceptron
- Recurrent networks: hopfield, boltzmann, hierarchical, long short term memory networks ...
- Self-organizing networks (e.g. Kohonen)

The in-depth description of these and other techniques goes beyond the scope of this document, and can be easily found in [3, 5, 4] or other textbooks on the subject of ANNs. As a generalization, it can be said that any ANN will implement a function $f(X) \in \mathbb{R}^n \to \mathbb{R}^z$ on the *input vector* $X \in \mathbb{R}^n$.

2.2.1 Classification output

One vs All method

In order to categorize the input data into multiple distinct classes, the most common solution is to use multiple binary outputs $Y' \in \mathbb{R}^z_{0..1}$ - one for each class - and to "select" the class with the highest output as the classification output.

Classification function

The classification function can be defined as $f_c(X) \in \mathbb{R}^n_{0..1} \to \mathbb{R}^z_{0..1}$ or $f_c(X) \in \mathbb{R}^n_{0..1} \to \mathbb{B}^z$ depending whether we are doing a *binary* classification by applying an output rounding function $c(y) \in \mathbb{R}_{0..1} \to \mathbb{B}$ to each input X_i. This output rounding function may have dependencies on the other rounding outputs, so it is sometimes best to think of a more generic classification function $C(Y) \in \mathbb{R}^z \to \mathbb{B}^z$, which selects the classification(s) that are predicted to be true, and can be applied at the end of all calculations. In the case of one vs. all classification, the rounding function is responsible for selecting only one class as true (or none if the output is not defined).

2.2.2 Cost function / Training Error

All Artificial Neural Networks (ANNs) have a specific *cost function* used to calculate the prediction error, when the real result is known. This depends on the structure of the ANN, but follows a common pattern. If, for each sample j of a set of these output/input pairs, z the number of outputs, m the number of training samples, X_j the vector of all the inputs to the ANN for sample j, Y_j the vector of expected outputs for that sample and $f(X) \in \mathbb{R}^n \to \mathbb{R}^z$ the network specific prediction function, then a generic cost (or error) function for a ANN on a single training example j can be defined as

$$E_j = \sum_{i=1}^{z} f(X_j)_i - Y_{ji}$$

and for the whole data set we have

$$E = \sum_{j=1}^{m} E_j = \sum_{j=1}^{m} \sum_{i=1}^{z} f(X_j)_i - Y_{ji}$$

[3] The Cost Function is also sometimes known as the Training Error or Energy Function, depending on the context and the type of network being used, and it is generally used as an input to the training algorithm which will then adapt the Neural Network improve learning on the current example.

2.2.3 Training and initialization

All ANNs will require a training phase that tries to minimize the error defined by the cost function over a given training set. This is achieved by modifying the values of the synapse weights of each neuron in the network according to a training specific *update rule*. For this reason each training method relates to a specific type or group of neural networks, and the end results are highly dependent on the initial weight values all through the network. For example, [11] explores in detail the importance of randomization vs the creation of local minima, and proposes what later became a widely accepted technique for choosing the initial weights for a multilayer network. [4, 5, 3, 37, 2] all provide overviews and details of several training methods for ANNs and discuss the problems regarding network initialization.

2.3 Performance

The measurement of the performance of a classifier is achieved by running the classifier on a set of samples of which the classification is already known. This set of samples could be part of, or the entire training set, but to avoid false results caused by bias in the classifier, it is recommended to use a separate *test* set. A common practice is to split the training set use 2/10 of the samples as the test set. Another 1/10 of the set can be used as a cross-validation set when necessary, or the test set can be extended to 3/10 of the available samples. The remaining 7/10 of the training set are used to train the learner. By comparing the classifier output with the correct classification on each sample of the set, we can extract four fundamental features:

- *tp*: The number of true positives, when the network predicts a correct classification
- *tn*: The number of true negatives, when the network correctly predicts the absence of a class
- *fp*: The number of false positives, when the network predicts a class that wasn't there
- *fn*: The number of false negatives, when the network misses the prediction of a class

[3] $f(X_j)$ could be further expanded to the function implemented by the specific ANN, but this would result in a loss of generality

2.3.1 Classification Error

Classification error is closely related to the training error, but is measured using a separate and independent test seti T of size r, that shares no data with the original training set, to prevent biased results (see "Bias and Variance"). If we consider, for each test set element j with Y_j being the expected output vector and θ_j the observed output vector, a measure of the error can be defined as [4] [5]:

$$\frac{\sum_{j=1}^{r} \sum_{i=1}^{n} (Y_{ji} - \theta_{ji})^2}{r}$$

This is an application of the Mean Squared Error (MSE) to the classification process.

2.3.2 Misclassification

Another, very naive and immediate measure of performance for a classifier could be to just count how many classifications have been performed correctly out of the total number of samples, in the form $\frac{Q}{n}$ where Q is the nnumber of samples classified correctly.

2.3.3 Accuracy

Accuracy is the simplest of the possible measurements, and represents the ability of a classifier to correctly make a prediction based on the current test set:

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn}$$

This measurement is subject to the bias of a *one-sided* sample distribution. To better explain the problem, we can imagine a test set where 99% of the samples have a negative expectation and a classifier that always predicts no class (a constant 0 output). The accuracy of this classifier would be:

$$Accuracy = \frac{0 + 0.99}{1} = 99\%$$

which is clearly a bad (albeit a corner case) measurement of the real performance.

2.3.4 Precision and Recall

In order to avoid this issue, Precision and Recall have been devised. These two metrics are very efficient at measuring performance of classifiers even in the case of a very *one-sided* sample distribution [31, 21, 47]. Precision can be thought of as the ratio of correct positive classification:

$$Precision = \frac{tp}{tp + fp}$$

[4] the error is normalized to the test set size r to ensure that errors are comparable when test sets have different sizes

[5] in this example outputs Y_i are of the type $\mathbb{R}_{0..1}$; to make this measure comparable in sigmoid-activated networks the result should be divided by 2

while Recall can be thought of as the ability of a classifier to find all the correct positive classifications in the dataset:

$$Recall = \frac{tp}{tp + fn}$$

2.3.5 F1-score

In order to understand correctly the performance of a classifier it is necessary to use both precision and recall. A common mistake is to accept a classifier for which only one of these two values is good. For this reason, rather than using two separate measures to be able to correctly measure the performance of a classifier, these features can be used to work out the F1-score (sometimes also called *F-score* or *F-measure*). This is the harmonic mean of precision and recall:

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

2.3.6 Bias and Variance

Bias and variance are two measures of how well the classifier is fitted to the problem. Adopting tr as the error on the training set and tt as the error on the test set:

- A high variance occurs when a classifier performs very well on the training set compared to the test set, therefore implying *overfitting*: $\frac{tt}{tr} >> 1$, tt is high, and tr is low.
- A high bias occurs when a classifier's performance is lowly correlated to both the training set and the test set. This means that both tr and tt are high. Paradoxically we can consider infinite bias in regards to a problem set to be equal to random performance.

[7] explores how training set size can affect bias and variance and reaches the interesting conclusion that although variance seems to decrease with the increase in size of the training set, bias seems to remain unaffected. [19] explores in detail the effects of bias and variance in the specific case of neural networks, and concludes:

> the bias/variance dilemma can be circumvented if one is willing to give up generality, that is, *purposefully* introduce bias. [...] one must ensure that the bias is in fact harmless *for the problem at hand*.

2.4 Weak learners

A weak learner is a learning algorithm with an error rate less than 50% - in other words a predictor that can do slightly better than making random guesses [24, 40]. Most of the time we will find that it easy to train a Neural Network to become a weak classifier without incurring the risk of biasing it towards the training set. [40] formally introduces the equivalence of strong and weak learnability, on top of which many of the concepts of ensemble learning are built.

Chapter 3

Ensembles

In a general sense, Ensembles or Committees are a way of combining *diverse* weak classifiers into a single, stronger classifier. This can be achieved in many ways, and we can identify two important sub-problems: *diversifying* the weak classifiers, and *combining* their outputs. For an in-depth review of classifier ensembles, please refer to [48, 40].

3.1 History

The first idea to combine a number of classifiers was exposed in [9]. This paper does not go much beyond the idea of mixing a linear classifier with a Neural Network, albeit it introduces a very powerful concept of combination and diversity. Undoubtedly, one of the pioneering documents on the subject is [40]. This piece of research introduces several new concepts and techniques:

- Weak learnability implies learnability. This means that no matter how weak a classifier, we can always construct a better one because the problem lends itself to be learned with arbitrarily low error.
- Combining multiple weak learners can lead to a stronger learner. This means that we have a use for weak classifiers, and they can be a powerful tool.
- Hypothesis boosting algorithm. This shows a way we can achieve the stated goals.

Schapire then continues his work and cooperates with Freund on a new variant of boosting, publishing several papers and creating what is now known as the AdaBoost algorithm [41, 42, 44]. Meanwhile, Breiman produces [8] where he introduces the concept of "bootstrap aggregating", which will later become known as bagging. Other significant works tackle the issue of combining the outputs of the weak classifiers, such as [25, 26, 16, 10], which introduce techniques like majority voting, max min and median rules and several others. Since these inital foundations have been laid, a large number of studies have been done, and I will cover more in the following sections, but the available literature is far too large to even be partially included in this document.

3.2 Overview

Because we can't consider any given ANN, no matter how well it is trained, as a perfect classifier, intuition will suggest that the opinion of more that one classifier should lead to a better result. However, if said classifiers are identical, there is no change in the result. To obtain a better performing classifier we need to obtain diversity in the output set. This concept can be easily understood by imagining a room full of people, who need to express an opinion about (classify) some data (an image, some text, a video, etc..). If there is only one person in the room, we are liable to all his mistakes. If there are many people we can imagine that if their opinions diverge, the collective will have a higher probability of getting to the correct conclusion, and they will be able to correct the output of those who were wrong. If, however all the subjects always reply unanimously, it would be the same as having only one person in the room. A postulate to this intuition would be that, instead of only considering weak classifiers ("all classifiers that do better than random"), we also consider bad classifiers ("classifiers that do worse than random"). These will then be considered with negative weight, therefore considering the inverse of their result. The important characteristic about ensembles is the the elements need to be negatively correlated, and they need to focus their error on different parts of the training set. This is generally achieved by introducing *diversity*. There are many techniques that developed and they focus on either diversifying the training set, the topology of the network, or in some cases both. Many techniques on how this diversity is recombined have also emerged over time, and the results vary widely. [48] contains a short proof of why a combination of weak learners will improve on a single learner:

> (it is) clear that we can reduce either the bias or the variance to reduce the neural network error. Unfortunately, it is found that for the individual ANN concerned, the bias is reduced at the cost of a large variance. However, the variance can be reduced by an ensemble of ANNs, leaving the bias unchanged.

[5] covers ensembles (which he calls committees) in the last chapter and introduces the concept with an example:

> [...] For instance, we might train L different models and then make predictions using the average of the predictions made by each model. Such combinations of models are sometimes called *committes*.

3.3 Training set diversity

We can diversify the training set in many ways:

3.3.1 Introduction of noise

The simplest way of creating many diverse training sets is to introduce random training data generated using the distribution of the original training data as a model. This allows to increase the size of the samples, from which to generate the small training sets. It is shown that although this technique reduces bias,

it is not increasing variance[35]. A very good example of this technique is given in the DECORATE (Diverse Ensemble Creation by Oppositional Relabeling of Artificial Training Examples) study in [33], where the introduction of artificial training samples based on the training set's distribution is shown to increase the generalisation ability of ensembles with small training sets.

3.3.2 Bagging

In bagging the idea is to create n training sets by taking an original training set of size m and picking m elements from it, with resampling, n times. This means that there will be overlap between the training sets (which keeps them correlated), but they will be diverse. This method can be made better if we ensure during the resampling of each set, that it is not equal to a previously existing set[8]. An example use of this methodology can be found in [34].

3.3.3 Boosting

The boosting technique also creates new training sets, but instead of randomly resampling like with Bagging, we instead run each classifier as soon as its training data is generated. We can then measure the performance of this classifier and use the results to retrain the next one to focus on the harder parts to learn. A well-known variant of boosting is adaptive boosting, or adaboost, where each set of data is assigned a probability. Those data whose prediction is similar to the training value receive an increasingly lower probability, and are therefore less likely to be selected[40][41]. An example usage of Boosting can also be found in [34].

3.4 Architectural diversity

Another way of diversifying the components of an ensemble would be to use different types of learners. This is not always guaranteed to provide negative correlation, but combined with differentiated training sets the likelyhood that each component will focus its error on a different part of the set becomes higher.

3.4.1 Diverse Neural Groups

If we pick several different implementations of neural networks (provided that we keep the same output range), we can use them as separate members of an ensemble. Diversity can be obtained by varying learning method, learning rate, network architecture (number of layers, number of neurons, etc), network type (self-organizing, perceptrons, etc) and network-specific parameters. An example of this broadly defined technique was used in [12], where the authors used evolutionary techniques to generate new emseble members. As long as the principle that these components perform *differently* than random and from each other is maintained, the contribution of each should be non-zero [1] [2]. If each

[1] It is possible that the contribution may be negative or zero if the aggregation method is not properly applied, but this requires adjustments in the aggregation rather than the diversification

[2] It is also possible that the contribution is so small to be approximable to zero. In such a case it would be better to discard the component and retry with different parameters in order

component network is visualized as an independent tree (although this may not be strictly true), the ensemble effectively becomes a forest of neurons. This is a simple technique and its results may not be as good as the others presented in this document. I plan to measure the performance and use it as a term of comparison to stacking.

3.5 Combining outputs

After managing to diversify all the different classifiers that make up the ensemble, the next step will be to combine these outputs in a way that will produce a single classification based on the results of the ensemble components. There are several ways of doing this and each technique has its own benefits[25]. We can initially differentiate between two strategies of combining components:

- Consider only a part of all the members of the ensemble
- Use all the components

This means that in the first case we need to always select the best performing candidates for each sample. It is helpful when classifiers have a "degree of certainty" for their classification (for example how close their output is to 0 or 1[3]), especially if some classifiers perform well on a small subset of examples.

3.5.1 Averaging

Taking the average of the outputs is quick, and is very helpful when the components have different local minima (thus are "diverse") in reducing the ensemble variance without increasing the bias by any significant amount[48]. A weight factor w can be added to each component x such that

$$\sum_{i=1}^{n} w_i = 1$$

which will allow to give each vote a different impact on the final result

$$Y = \sum_{i=1}^{n} w_i x_i$$

It may be observed that this is the same function as the weighted sum of inputs in a neuron. This could be taken even further by applying the sigmoid function $P(t)$ defined in chapter 1, making this effectively another artificial neuron that could be trained. The constraint on the sum of the weights would no longer apply. This could be imagined as the simplest version possible of Stacking (see further).

to maximize the use of computational resources

[3]We could formalise this by sorting in descending order for $\chi = (0.5 - y)^2$

3.5.2 Majority voting

When using majority voting we consider the classification that was the output of the highest number of classifiers. It is easy to see why it is called "voting" and it is very straightforward to implement and understand, and executes very quickly. A concrete discussion of the usage of the many majority voting variants available is done in [17].

3.5.3 Stacking

In stacking we can take all the single weak classifiers we created using any of the previous techniques, and use them as input for a subsequent classifier. This requires yet another split of the training set, such that a subset is left that has not been used to train the *first space* classifiers. This new subset will be used to train the *second space* classifier[13]. This second dataset will have to be modified to match the second space classifier's inputs. We combine the outputs of all the first space classifiers based on the dataset's input, and use them as input values. The expected output remains unchanged.

3.5.4 Ranking

In order to implement ranking we need to make an essential modification to the underlying classifiers, so that instead of providing a single classification option they will return a list of resulting classes ordered by relevance. There will be a selecting classifier that will look at the results of these ranked classifications and choose the most suitable result [23][1]

Chapter 4

The Encog Library

Encog is an open source library maintained principally by Heaton Research, and it implements various techniques for machine learning and artificial intelligence. It was initially developed in Java, but it has also been ported to many other languages. For the scope of the project we will focus only on the Java implementation. The Java implementation of Encog is hosted on github at `https://github.com/encog/encog-java-core` and is released under the Apache license, which makes it very easy to work with and modify.

4.1 Architecture

Encog is structured as a single library, with many packages. Unfortunately at the time of writing no official UML diagrams were available for the Java implementation. Using UMLGraph to generate diagrams from the code reveals that the complexity of the code is too high for the possibility of having a simple significant diagram to include in this documentation. I have however separated some portions of the library and tried to achieve an "overall" view by describing separate sections.

4.1.1 Machine Learning Packages

Encog includes the interfaces and classes for ML under org.encog.ml. For the purpose of this document, I have included only the principal interfaces in the following diagram.

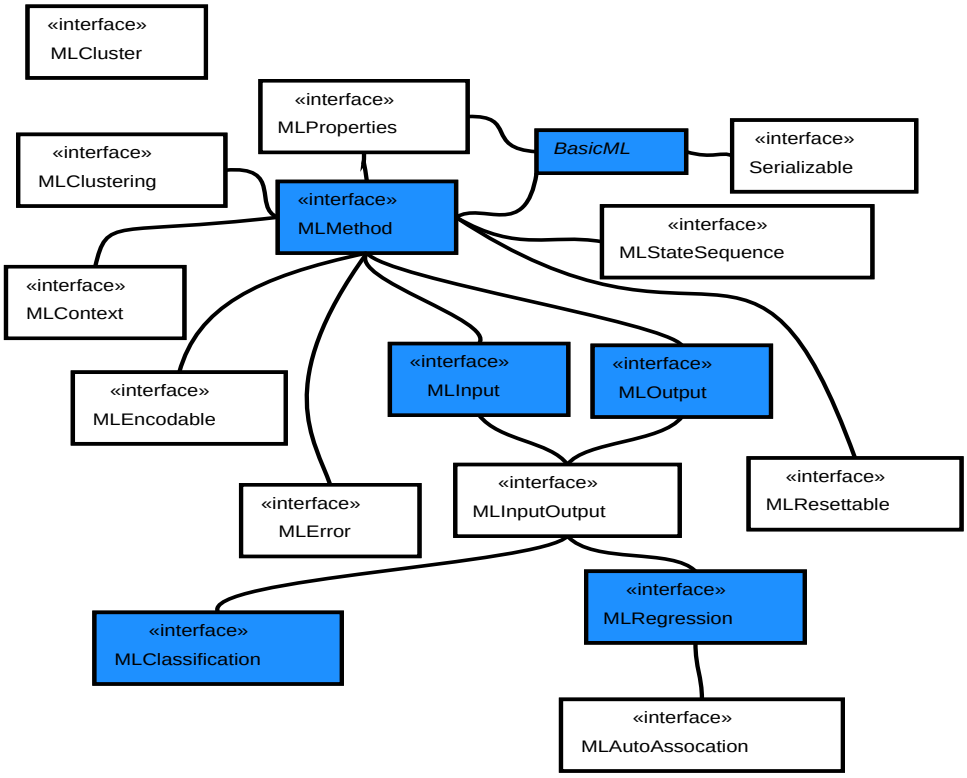


Figure 4.1: Basic encog ML package

These interfaces are used in most sections of Encog, either for the definition of inputs and output (*MLInput* and *MLOutput*), to define a type of ML method (*MLRegression* and *MLClassification*) and other common properties that are easily defined via inheritance, and that provide compatibility between similar objects.

4.1.2 Neural Network Packages

In org.encog.neural we can find the neural network specific implementations. Most of these are implementations of ML interfaces, which is why I decided to implement the ensemble techniques based on ML interfaces rather than Neural interfaces. The classes and interface that are most relevant to this project have been highlighted in light blue.

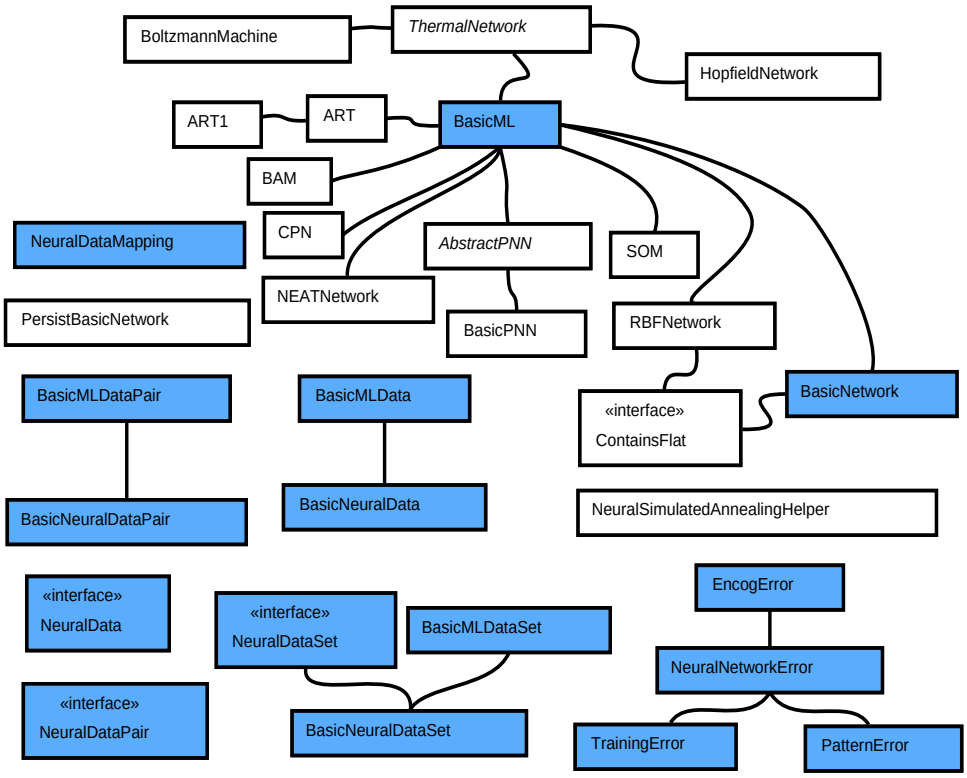


Figure 4.2: Common NN classes and interfaces in org.encog.neural

These classes provide the basic NN implementation. *BasicNetwork* provides a simple multi layer perceptron implementation that uses the *NeuralData* interface for input and output. *NeuralDataSet* provides an interface to define datasets such as training sets or test sets. *NeuralDataPair* is an interface that contains one input datum and one output datum, making it suitable for supervised learning. The *Error* family of classes provide an abstraction for representing errors inside Encog.

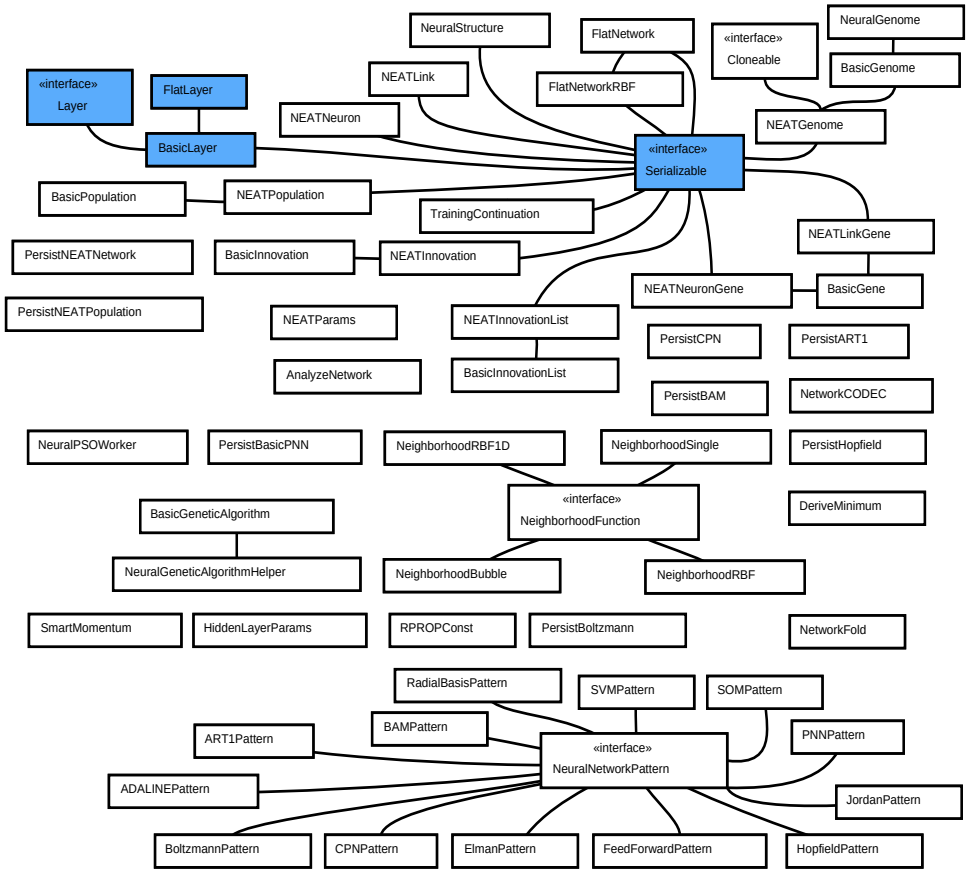


Figure 4.3: NN architecture related classes and interfaces in org.encog.neural

Most of these classes implement features for more advanced NN algorithms that haven't been used in this project, with the exception of the *Layer* family, which implement single layers inside a perceptron.

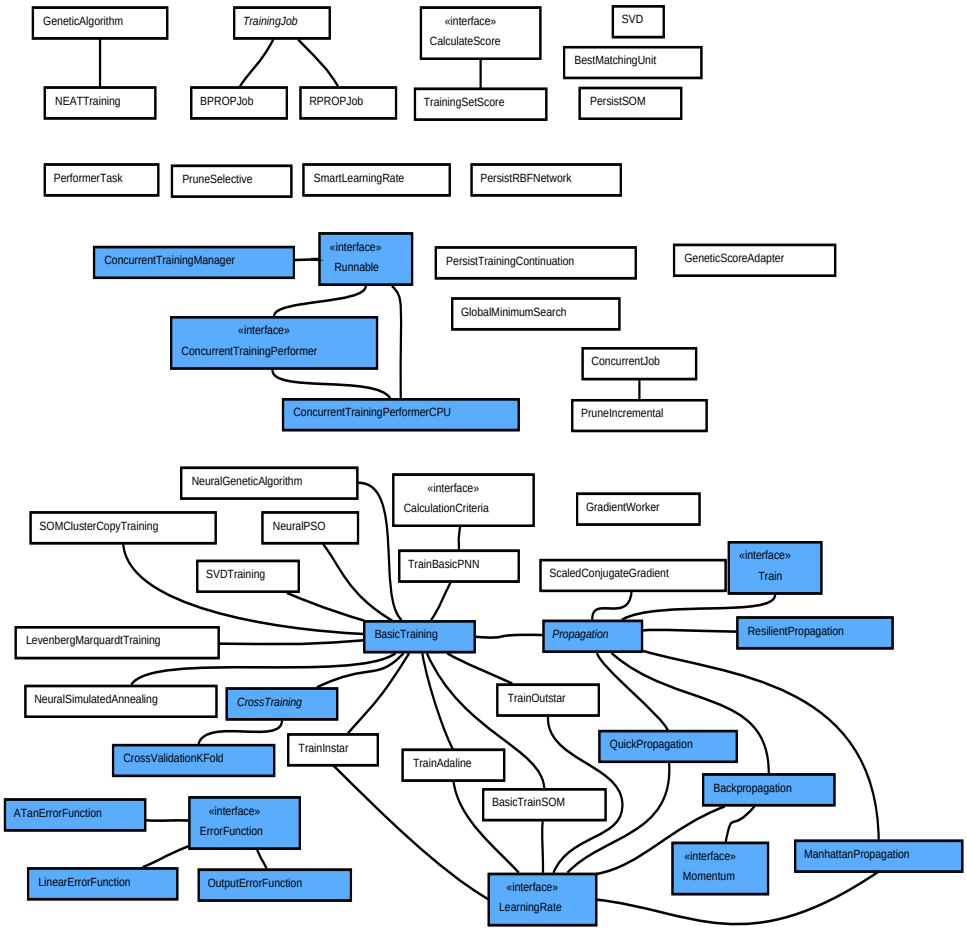


Figure 4.4: NN training interfaces and classes in org.encog.neural

These classes and interfaces provide an implementation of training algorithms, of which many haven't been used. The name of each class represents which specific algorithm it implements, and in this specific project we have focused only on the *Propagation* family of classes. The *ErrorFunction* interface represents the function used to calculate error in the training exercise, and the *Runnable* interface and its descendants provide support for parallel training.

4.1.3 Activation Functions

The activation functions have their own separate package called org.encog.engine.network.activation. This is to allow reuse of these activation functions in other machine learning techniques.

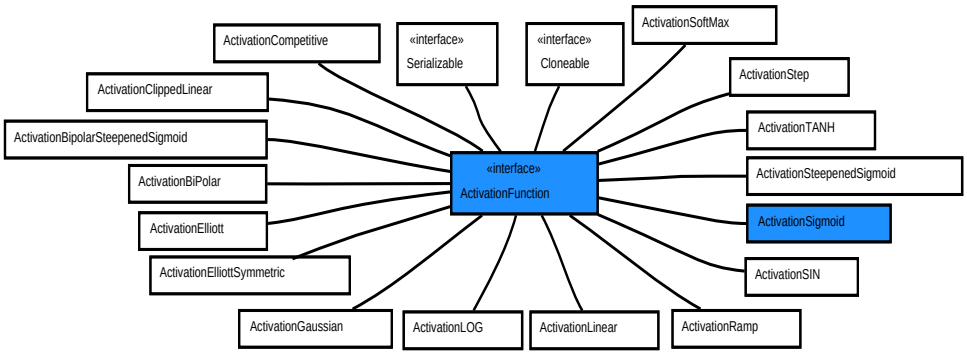


Figure 4.5: NN activation functions

The only activation function used in this project is the sigmoid activation $P(t)$ presented in chapter 2, which is implemented in *ActivationSigmoid*.

4.1.4 Utility Packages

Encog needs a large number of additional packages to deal with object persistence, loading of files and in these extra packages we can find many helpful tools, but they are not strictly relevant to the project implementation.

Chapter 5

Implementation

5.1 Layout of the project

The code section of this project can be divided in two parts: one part relates to the code changes to the Encog library itself, while the other relates to the software written to obtain performance data. In terms of codebase size, it is an interesting observation that more code was needed to implement the tests than to implement the changes to Encog.

5.1.1 Packages

The following Java packages were created for benchmarking:

- ensembles
- helpers
- helpers.datasets
- perceptron
- techniques

and the following were added to Encog:

- org.encog.ensemble
- org.encog.ensemble.adaboost
- org.encog.ensemble.aggregator
- org.encog.ensemble.bagging
- org.encog.ensemble.data
- org.encog.ensemble.data.factories
- org.encog.ensemble.ml.mlp.factories
- org.encog.ensemble.training

5.2 Adding features to Encog

This section covers all the components that have added to Encog, at a medium-to-high level. For more implementation details, the reader is deferred to the javadoc documentation generated or the source code itself.

5.2.1 Design decisions

One of the most interesting decisions was to change most interfaces to be abstract classes. This happened fairly late during the implementation, but reduced the codebase considerably, and eliminated most of the repeated code. For this reason *interface* can be used interchangeably with *abstract class* throughout this whole chapter. The decision to delegate responsibility to other interfaces was also good, as it permitted easy implementation of new functionality in an almost completely pluggable fashion [1]

5.2.2 Architectural changes

I have made a heavy use of Factory patterns and inheritance during the design process. This has allowed me to make the software structure very modular, and adding new implementations at a later date has revealed itself to be a simple task from an architectural point of view. I tried to share as many interfaces with the already existing parts of Encog as possible, although some had to be aggregated, as with *EnsembleML*.

5.2.3 Class design model

In this section the various components are presented and their functionality explained. They are separated by the interface they implement, which can be associated with functionality groups.

Ensemble classes

This interface represents the main ensemble. It contains fields for all the other interfaces that are needed to implement complete ensemble functionality. The method of constructing a complete ensemble requires varied steps, depending on the required characteristics. As a result of this, it lends itself well for being used in a Builder pattern [18]. The generic model of this class type is to use an EnsembleDataSetFactory to create one new EnsembleDataSet for each member and to create the members themselves using an EnsembleMLMethodFactory, which also requires a training method. Aggregation is done with an EnsembleAggregator and the training of the members is done using an EnsembleTrainFactory, both passed at instantiation time.

Bagging This class implements a version of Bagging. It uses the ResamplingDataSetFactory class to create the bootstrapped datasets which will then be passed to whichever EnsembleMLMethodFactory is used to create the ensemble members.

[1] minor adjustments had to be made to the interfaces early on to permit some of the new algorithms to fit in, but at a later stage it wasn't necessary to modify the new interfaces any more

AdaBoost This class implements the standard AdaBoost algorithm and uses a WeightedResamplingDataSetFactory to generate EnsembleDataSets that resample based on probability weights that are calculated at the training of each member. This method is particular because all the members have to be trained sequentially.

Algorithm 1 Training in AdaBoost

```
procedure TRAIN(ANN,TargetError)                                  ▷
    Initialize all members of D to 1/n                ▷ D is the weight vector
    for all i members do
        Get a new training dataset
        Create member i using the training factory and new dataset
        Get weighted error for new member
        Update member weights vector
        Add member to main list
        Update D with new values
    end for
end procedure
```

Stacking This is the implementation of Stacking, also known as Stacked Generalization. Although the implementation allows for any EnsembleAggregator to be used, it would not be a correct implementation if it didn't use an instance of Metaclassifier. Unfortunately there isn't a way to impede the use of other EnsembleAggregators without changing the Ensemble interface. This could be achieved with a language which easily supports type variants or runtime type mathing. Scala, Haskell and OCaml are all examples of such languages.

EnsembleAggregator classes

This family of classes implements the aggregation of the results via the evaluate function. This takes a list of computed outputs and performs the specific aggregation function, returning a new output of the same cardinality with the final decision.

Averaging This is a simple aggregator that returns the average of all the outputs for each class.

MajorityVoting This is another simple aggregator that first thresholds the outputs to put them in binary format, and then picks the output with the highest vote count.

MetaClassifier This aggregator takes an EnsembleMLMethodFactory and an EnsembleTrainFactory at instantiation time, creates a classifier (the "tier 2 classifier" from Stacking), and trains it on a special training set that is generated by aggregating the outputs of all classifiers for each instance of the original training set. The final output is then the output of this classifier when presented with the ensemble member outputs.

EnsembleDataSetFactory

These classes generate data sets for training ensemble members, by implementing the getNewDataSet function, which returns a new dataset of size k, specified at initialization. An initialization dataset will have to be given, from which the data instances are picked.

ResamplingDataSetFactory This factory will generate datasets of size k, by randomly picking entries from the initalization data set, with resampling.

WeightedResamplingDataSetFactory This factory will return data by randomly picking entries from the initialization data set, with resampling, and by taking account of a weight vector U.

WrappingNonResamplingDataSetFactory This factory will always return the k entries between $k*i$ and $k*(i+1)$, with i being the count of times that getNewDataSet has been called before. If the end of the initialization dataset is reached, then the search wraps back to the beginning ad infinitum.

EnsembleML classes

The EnsembleML interface adds functionality on top of the MLMethod class in Encog, and implements the function of an ensemble member. The reason for having this interface is to allow new sub-types of this class to be implemented transparently to the current code, to allow new techniques to be implemented.

GenericEnsembleML This is the generic implementation of an EnsembleML. No other implementations were necessary at this stage, but it did not seem appropriate to remove the generality provided by having the separate interface.

EnsembleMLMethodFactory classes

These factories generate ensemble members of type EnsembleML, and require an EnsembleTrainFactory and an EnsembleDataSetFactory to operate.

MultiLayerPerceptronFactory This factory create multi-layer perceptrons by taking a specification of activation function and layers, under the form of a list of integers.

EnsembleTrainFactory

This family of factories will create the training methods (of type MLTrain) that will be used in EnsembleML.

BackpropagationFactory This class implements regular backpropagation [3]

LevenbergMarquardtFactory This class implements LMA [29]

ManhattanPropagationFactory This class implements Manhattan propagation [36]

ResilientPropagationFactory This class implements Resilient propagation [36]

ScaledConjugateGradientFactory This class implements the Scaled Conjugate Gradient algorithm [32]

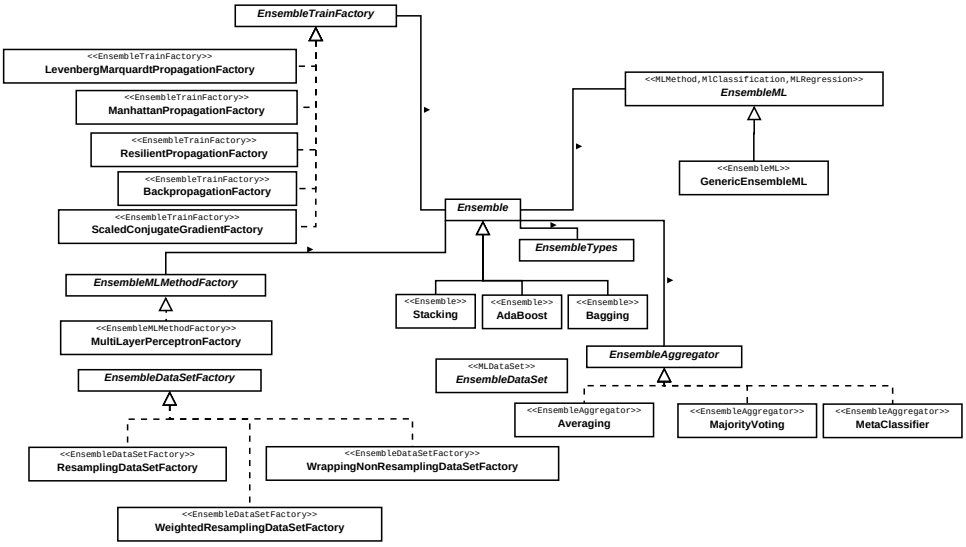


Figure 5.1: Class diagram of added components

5.3 Creating a framework for testing

A challenge has certainly been the benchmarking of the new techniques. The implementation has required the creation of a framework, albeit very small, for describing and loading datasets, applying an ensemble learner, and extracting results. This is without considering all the necessary tools to actually run the benchmarks.

5.4 Source management

In order to manage such a complex codebase, it was necessary to adopt some measure to store revisions, source distribution between hosts (also for backups), and automate building and testing.

5.4.1 Revision control

The source code for Encog is hosted on github (https://github.com/encog), so the straightforward solution to managing all the Java code for this project was

to use git. This should simplify the patch submission process. All other files related to the project (including the project proposal and this report) are stored in a separate mercurial repository. The main reason for choosing mercurial is its simpler merging mechanism and approach to multiple heads. A convenient side-effect of this choice is that it is impossible to accidentally attempt to merge the project-related repository with the encog repository.

5.4.2 Automated building

In order to use up-to-date code in the simulations, some sort of automated compilation and building of the sources was required. I opted for an ant build script because it had the lowest requirements and was the easiest method to implement quickly. This was enough to make sure that building the sources for the jobs did not require a full recompilation at every job submission.

Chapter 6

Benchmarking the new features

6.1 Running tests

In order to obtain data to show the efficacy of the various ensemble approaches, it has been necessary to run the test program on a variety of datasets and collect the data on performance in respect to the ensemble size, collect the data, and represent it in a graphical way that highlights the immediate validity of the techniques under test.

6.1.1 The test plan

For each dataset, the plan is to find some semi-optimal single-perceptron parameters, and try to improve on the performance of that perceptron by applying, in turn, all the ensemble techniques, with all the member output aggregation methods, in a linear search for the optimal number of ensemble components (this has been limited between 1 and 300, mostly due to time limitation). This generates a series of results that can be graphed in respect to the ensemble size.

6.1.2 The test program

The test program is a relatively simple piece of code that will read a predefined problem description file, take input parameters on how to create and train an ensemble to learn the dataset, and evaluate the performance of the ensemble on both the training data and a dynamically determined test set.

Arguments

All the command line arguments to the testing tool are in one way or another related to the definition of which technique and which parameters to use to create and train the ensemble. In order, they are:

- technique: the ensemble technique to use for this run (bagging, adaboost)
- problem: the problem set description file

- sizes: comma separated list of ensemble sizes to try
- dataSetSizes: comma separated list of sizes for generated ensemble training sets
- trainingErrors: comma separated list of target training errors
- trainingSetSize: size of the original training set (used to generate the ensemble training sets)
- activationThreshold: the minimum value for an output to be considered 1
- training: training algorithm (rprop, backprop, scg)
- membertypes: description of the ensemble members architecture in the form method:parameters:activation (e.g. mlp:100:sigmoid)
- aggregator: the method used to aggregate the outputs of each ensemble component (majorityvoting, averaging)
- verbose: whether to print out all intermediate training values (useful for plotting "training curves")

Dataset description

In order to make it easy to load in different datasets, I have made the dataset description an external configuration file. This contains all the values needed to parse the dataset files correctly and to perform one-vs-all classification in the correct manner on the specified dataset. The descriptor file contains entries in the ¡setting¿=¡value¿ format:

- label: what label to use in the output, to identify this dataset.
- outpus: how many output classes there are
- inputs: how many input classes there are
- read_inputs: how many columns contain the classification values for the training data
- inputs_reversed: if true, the input classification is before the input values
- mapper_type: what mapping to use for the classification representation
- data_file: the path of the CSV containing the training set data

Output

The output is in CSV format on stdout at execution time, and contains the following fields, in order:

- the label of the problem set
- what set this is (test/training)
- the ensemble technique used in this run

- the training method used
- the type of ensemble members
- the layout of the ensemble members
- the aggregator used to decide the output
- the size of the ensemble
- the size of the data set
- the target training error
- the misclassification
- microaveraged accuracy
- microaveraged precision
- microaveraged recall
- microaveraged F1
- macroaveraged accuracy
- macroaceraged precision
- macroaveraged recall
- macroaveraged F1

6.2 Infrastructure layout

6.2.1 The PBS system

In order to efficiently run several test jobs with so many different parameters, it has revealed itself convenient to use a batch system. The system of choice is TORQUE (http://www.adaptivecomputing.com/products/open-source/torque). It is one of the many derivatives of the original OpenPBS and is reasonably straightforward to setup and manage. In order to preserve security when running a PBS system distributed across several physical locations, I have opted for routing all PBS traffic over an Openvpn network. It has revealed itself necessary to write a few patches for TORQUE, which have been submitted back to the community for review.

6.2.2 AWS compute nodes

The large number of simulations, and the necessity to be able to re-run some tests towards the end of the project, have made it necessary to create a system that was flexible in terms of size. Amazon Web Services has served well as an enabler of elastic instances that could run these jobs "out of the box". A node image has been devised, such that at first boot it would configure itself to join a virtual private network, connect to the master PBS node, and start executing jobs instantly. At peak times there have been 25 compute nodes, and the ease with which it was possible to turn them on and off at will has undoubtedly simplified the execution part of the project by a great deal.

6.2.3 Submission tools

To make submission to the PBS system possible I had to write a set of scripts that would manage output handling, ensure that the test application was up to date at run time, and speed up creation of new job scripts. *run_job* contains all the basic functions, but the real job wrapper scripts are named by the job type they represent. All submissions are made in the form of job arrays [1], and the job index is used as the ensemble size for that specific benchmark run.

Environment

Scripts like run-environment.sh are responsible for updating the source tree from git, compiling all the framework using ant, and creating aliases and shortcuts to make job definition more standardised. These scripts and the ant *build.xml* definition file live with the framework source code in *encog-checkout/src/runs*.

Job Scripts

The job scripts are responsible for calling out to the environment, setting up input data where necessary, and submitting the output data to the mercurial repository. Examples of these are in the *batch_jobs* directory.

6.2.4 Aggregating results

All results are committed into the *projects/mscproject/runs_output* subdirectory of the mercurial repository. *generate_csv.sh* creates a csv with all the results, which facilitates importing into a worksheet program like oocalc or Excel.

6.2.5 Creating graphs and tables

In order to create all the graphs and tables contained in this report, a tool of modest complexity was needed, so that the process could be automated and re-run at several stages of the project. *make_graphs_and_tables.pl* is a Perl script that takes all the necessary steps to create the material, and deposits it in the correct place for it to be included in the report compilation, done by *build.sh*.

6.3 The data sets

All the real-life datasets used for testing are taken from the UCI machine learning repository (http://archive.ics.uci.edu/ml/). I have also used a tool called datasetgenerator (http://code.google.com/p/datasetgenerator/) to generate exact datasets, to verify the algorithms. All datasets have been split in training and test sets (as per the discussion in 2.3), and the performance has been measured for both sets in all cases, so to facilitate the spotting of overfitting. The measure of performance that seems more appropriate when looking at the output in this case seems to be misclassification, as defined in 2.3, as it represents the "number of mistakes", which can be easily understood. The size of each training set and test set has been predefined, but the selection of which instances make

[1] In a PBS system, a job array is a shortcut for submitting multiple copies of the same job, and each one receives an index number at execution through the environment.

it into the training set is a random process that happens in runtime, and has therefore different results every time. The remaining instances are used in the test set. The reason for doing this is to avoid giving an unexpected advantage to any specific technique. These are all the dataset (generated and non) that have been used:

6.3.1 Haberman's Survival

This is the data resulting from a study on the survival rate of patients after receiving surgical treatment for breast cancer

- URL: http://archive.ics.uci.edu/ml/datasets/Haberman's+Survival
- Relevant papers: [22, 27, 30]
- Data instances: 306
- Training set size: 200
- Inputs: 3
- Output classes: 2, in numeric format (1 field)

6.3.2 Letter Recognition

This dataset contains a number of statistical and edge properties of some characters and the purpose of the classification is to establish which letter of the alphabet is represented in the rectangular section of pixels described by these properties.

- URL: http://archive.ics.uci.edu/ml/datasets/Letter+Recognition
- Relevant papers: [15]
- Data instances: 20000
- Training set size: 16000
- Inputs: 16
- Output classes: 26, in letter format (1 field)

6.3.3 Landsat

In this dataset there is a small section of a scan from the NASA Landsat satellite, with each line representing a 3x3 pixel area, with readings in 4 spectral bands. The goal is to classify the type of soil contained in the pixel area. This dataset has been discussed by many authors, including Schapire and Freund themselves while discussing Boosting algorithms.

- URL: http://archive.ics.uci.edu/ml/datasets/Letter+Recognition
- Relevant papers: [43, 45, 20, 46] and many more, referenced on the dataset's web page.

- Data instances: 6435
- Training set size: 5000
- Inputs: 36
- Output classes: 7, in numeric format (1 field)

6.3.4 Magic

This dataset contains generated data that simulates high energy gamma particle observations in an atmospheric Cherenkov telescope. I have chosen this dataset beacuse it has a large number of instances and given that it is a generated dataset it should be easier to find a working machine learning technique to model the set.

- URL: http://archive.ics.uci.edu/ml/datasets/MAGIC+Gamma+Telescope
- Relevant papers: [14, 39, 6]
- Data instances: 19020
- Training set size: 16000
- Inputs: 11
- Output classes: 2, in numeric format (1 field)

6.4 Reducing the search space

Because of the large number of options available for tweaking, the search space for optimal settings for the test runs is very large. A few "tricks" have been necessary in order to reduce both the dimensionality of this space and the bounds of each dimension searched.

6.4.1 Previous literature

In order to improve the effectiveness of the search for optimal parameters, I have tried to extract ideas from previous literature on the specific datasets (where applicable). This helped pick good starting points. All literature that applies to a specific dataset is mentioned in that dataset's specific section.

6.4.2 Picking parameters

In order to reduce the search space even further, I have extracted empirically good parameters for the single classifier example, for each of the problems. This involved plotting "learning curves" for each neural architecture, which plot the MSE in relation to the training iteration, as well as the training and test set misclassification, at each step of the training process. The most convenient way is to use the existing framework, with bagging-1 and averaging. This should be equivalent to running a NN classifier on its own.

Ensemble Member Type

The main goal of creating these "learning curves" is to be able to pick an ensemble component type for each problem without having to run benchmarks across the whole space of possible member types. We already know by the limitation set out before commencing the exercise that the only members will be multi layer perceptrons (although they may not be optimal learners for these problems). The space of how many neurons, the training method, the training target error still need to be searched.

Training Method The space searched for training methods contains mostly propagation-type trainers, and includes Resilient Propagation, Back Propagation, Manhattan Update Rule, Scaled Conjugate Gradient and Levenberg-Marquardt. It turns out that for these problems, using only simple multi-layer perceptrons, the best option is to use Resilient Propagation in all cases. It is reasonably quick and provides decent results.

Target Training Error All propagation training in Encog requires a "target" internal mean-squared error (MSE) to which the ANN is trained. The basic algorithm is as follows:

Algorithm 2 Encog basic propagation algorithm

```
procedure TRAIN(ANN,TargetError) ▷ Train an ANN to this TargetError
    while getError(ANN) ¿ TargetError do
        PerformTrainingIteration(ANN)
    end while
end procedure
```

6.4.3 Training Curves and Final Parameters

Haberman

Here we can see that the test with 30 neurons shows a much better MSE, which reaches and stays below 0.17 very quickly. Misclassification errors are also more stable than the other examples with 100 neurons and above, which have high noise levels on the test set, indicating probable overfitting.

- Architecture: Multi layer perceptron, 1 middle layer of 30 neurons
- Target training MSE: 0.17
- Ensemble size range to test: 1-300

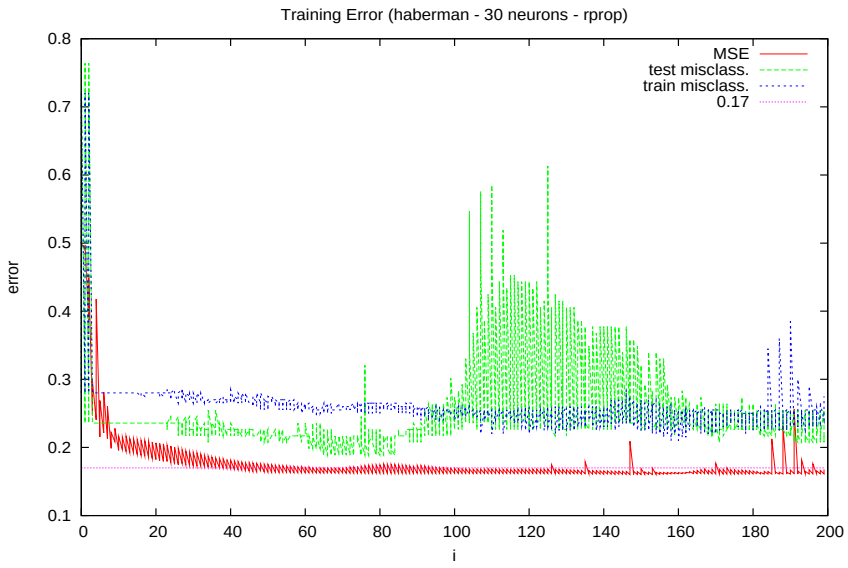


Figure 6.1: Training and test error for Haberman with 30 neurons

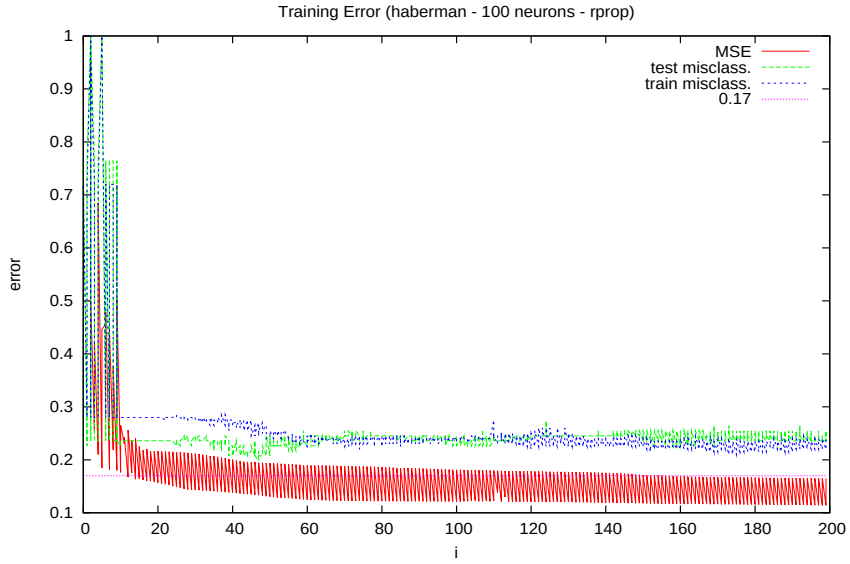


Figure 6.2: Training and test error for Haberman with 100 neurons

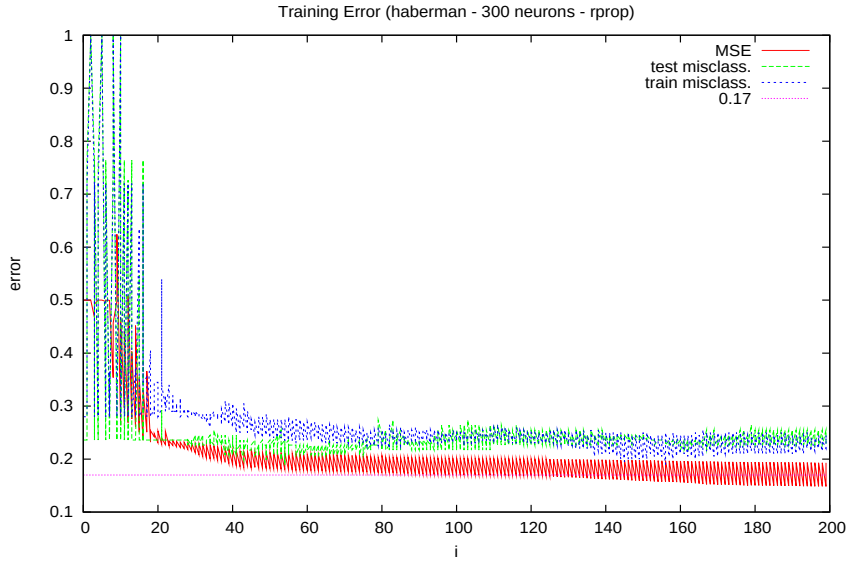


Figure 6.3: Training and test error for Haberman with 300 neurons

Letter Recognition

This problem set seems to have an MSE that descends quickly below 0.05 and then moves very slowly after the first few training iterations. Noise levels are very low for the first 500 iterations for both 100 and 300 neurons, while anything below 100 does not seem to have enough capacity to learn all the patterns. We seem to be gaining more correct result as the training continues, although the noise levels that appear at around the 650th iteration may indicate that that is the point where we start overfitting the data.

- Architecture: Multi layer perceptron, 1 middle layer of 300 neurons
- Target training MSE: 0.015
- Ensemble size range to test: 1-300

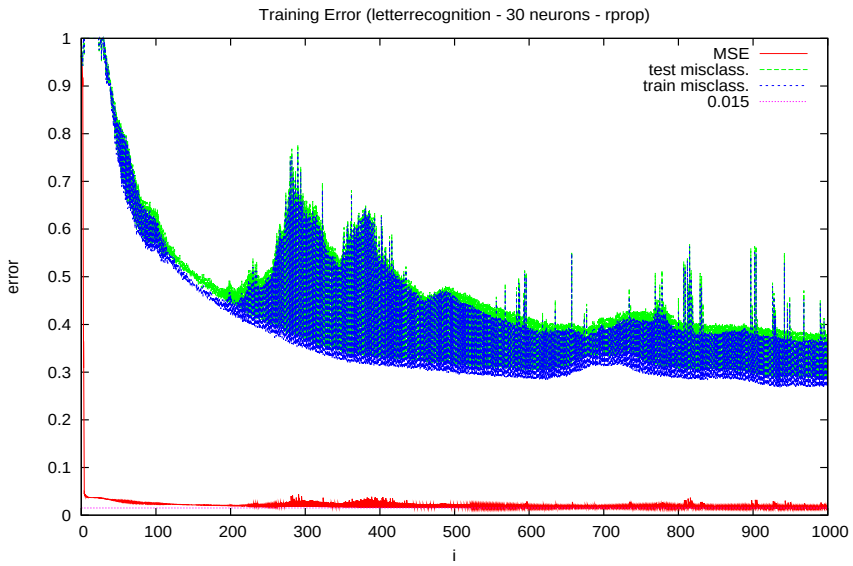


Figure 6.4: Training and test error for Letter Recognition with 30 neurons

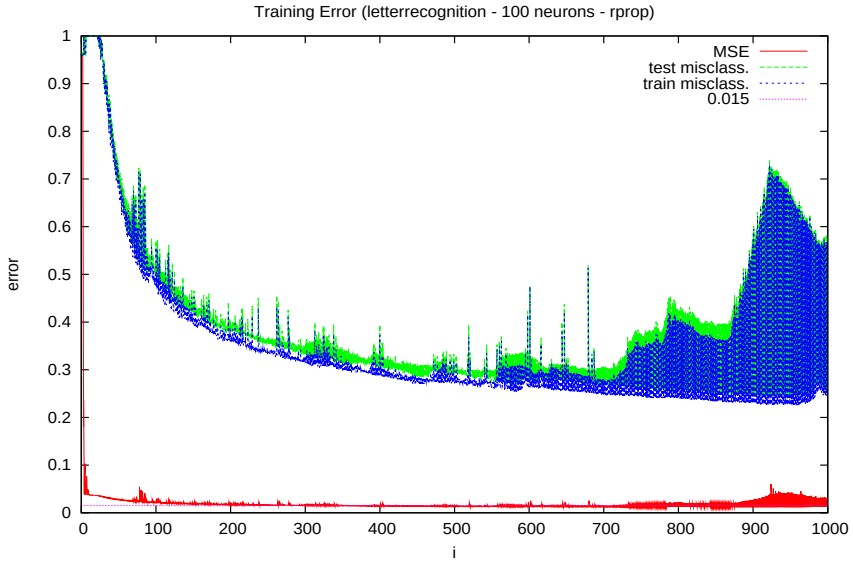


Figure 6.5: Training and test error for Letter Recognition with 100 neurons

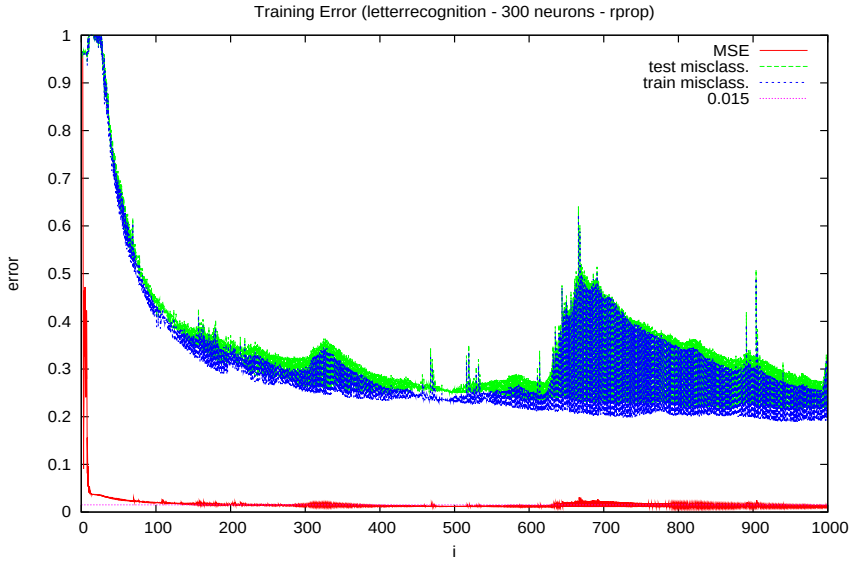


Figure 6.6: Training and test error for Letter Recognition with 300 neurons

Landsat

We see some large amounts of noise here at all neuron counts. This may be an indication that multi-layer perceptrons could not be the ideal learning algorithm for this data set. Nevertheless we see that at 300 neurons the noise is much smaller and at least we should be able to obtain some better-than-random results, which is all we need to benchmark ensemble techniques.

- Architecture: Multi layer perceptron, 1 middle layer of 300 neurons
- Target training MSE: 0.05
- Ensemble size range to test: 1-3000

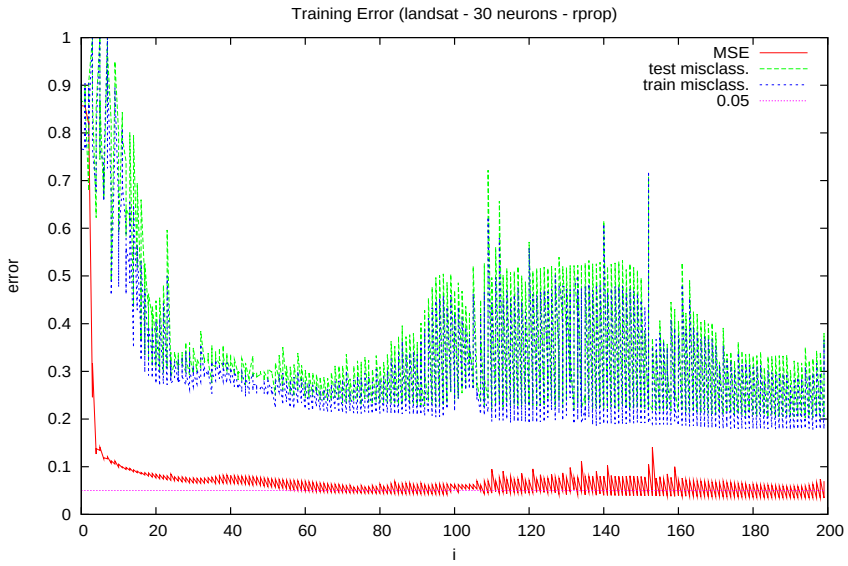


Figure 6.7: Training and test error for Landsat with 30 neurons

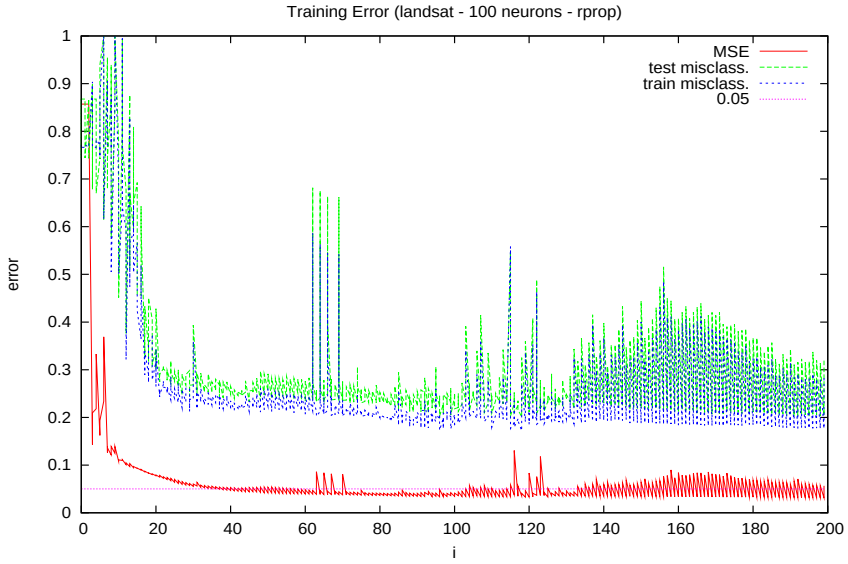


Figure 6.8: Training and test error for Landsat with 100 neurons

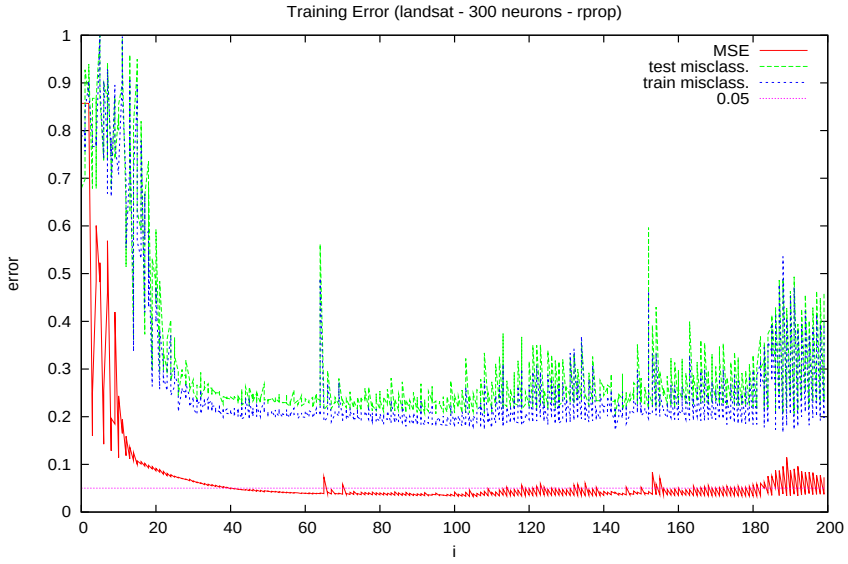


Figure 6.9: Training and test error for Landsat with 300 neurons

Magic

The MAGIC Telescope dataset appears to be a hard set to generalize for a multi-layer perceptron. The performance at 100 and 300 neurons is almost identical, and the noise level is very low. With these parameters being very similar, the assumption is that the cheapest of the two options is better.

- Architecture: Multi layer perceptron, 1 middle layer of 100 neurons
- Target training MSE: 0.10
- Ensemble size range to test: 1-3000

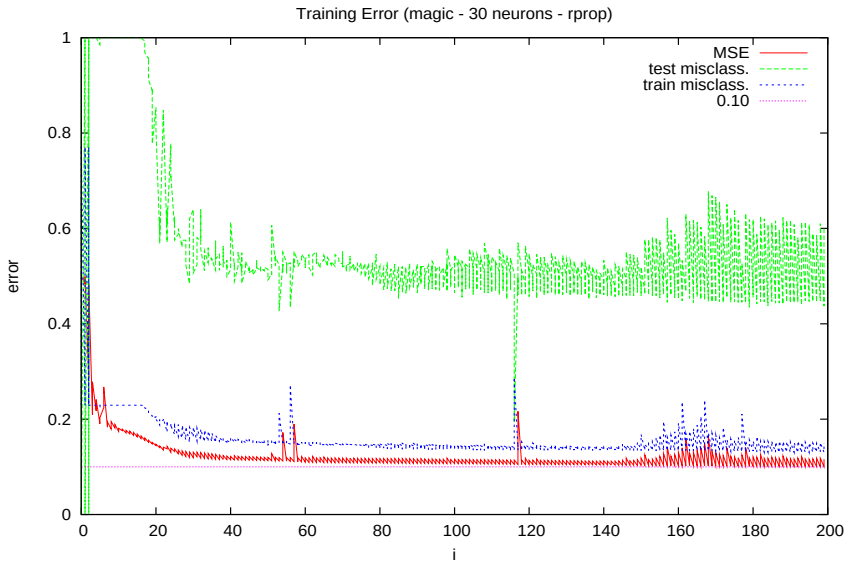


Figure 6.10: Training and test error for Magic with 30 neurons

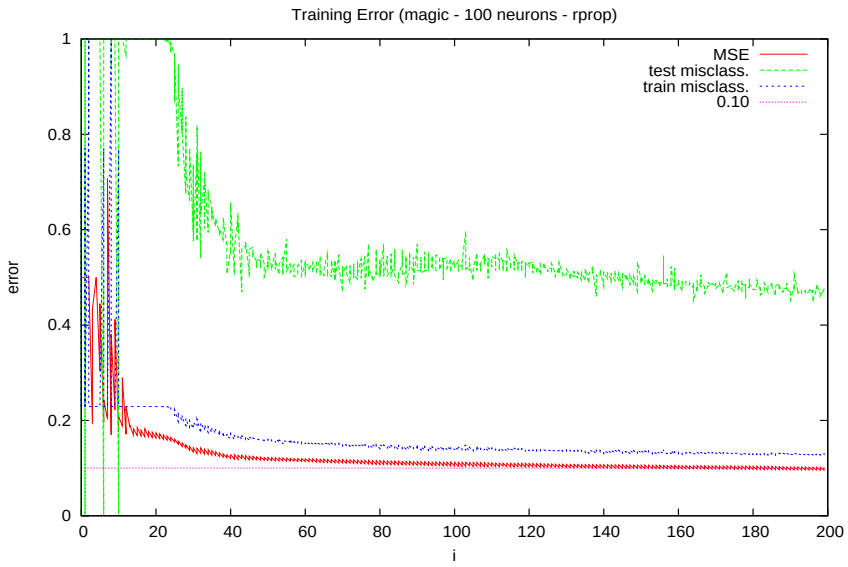


Figure 6.11: Training and test error for Magic with 100 neurons

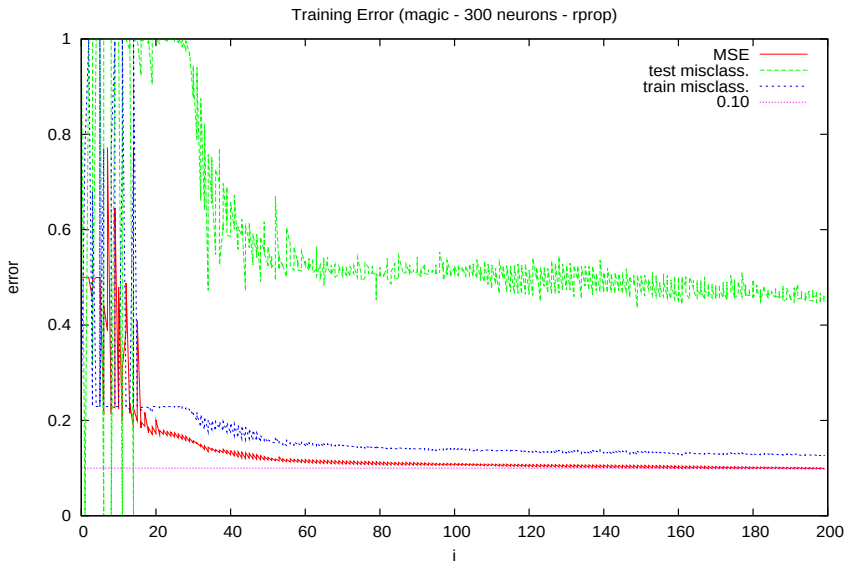


Figure 6.12: Training and test error for Magic with 300 neurons

6.4.4 Dividing ranges

In order to improve the time required to run these searches, I have found it useful to divide each parameter's space into smaller ranges, so that several separate jobs could be run at once. This made the search quicker without having to reduce the search space itself, at the cost of utilizing more hardware.

Chapter 7

Test results

In this chapter the raw results collected from the benchmarks are presented. For each possible combination of ensemeble technique, voting mechanism and problem set the following data have been generated:

1. a commentary of the data obtained from the benchmark
2. a table illustrating the average of the performance values for both the test and training datasets for each ensemble size [1]
3. a graph illustrating the performance measures for the test set, in relation to the ensemble size
4. a graph illustrating the performance measures for the training set, in relation to the ensemble size
5. a per-class accuracy histogram for each ensemble size
6. further considerations, if any

For succintness, only the data information worthy of commentary has been included in this chapter, while all the other generated graphs can be found in the data_plots/final directory of the project repository.

Each benchmark has been run a minimum of 3 times, to ensure that the data is less fallible to outliers. Characteristic traits that we look out for in the data are:

- *Measurement improvements.* For all the measure we are looking for indication that increasing the ensemble size produces better results. Ideally this is represented by local minima (or maximums) in the measurements.
- *Optimal ensemble sizes (or ranges).* We are looking for ranges on the x axis where the performance measurements adopt lower values. Ideally the local minima from the previous characteristic lie in this region.
- *Noise reduction.* We are looking for ranges where the variance in the performance measurements is reduced.

[1] The term "ensemble size" refers to the number of members of the ensemble used for a specific benchmark

- *Class bias.* It is possible that a classifier may be performing better for certain classes, so we extract a histogram of accuracy on each class for the training set. If there is a class that is clearly outperforming or underperforming we may have class bias. This is an especially well-known problem with multilayer perceptrons, and there are training techniques to remedy this [28].

7.1 Bagging

7.1.1 Majority voting

Haberman

It would seem that in this case we get a stable, good performance at around 7 ensemble members. After this the measurements do not decline heavily, but the variance in the results does not seem to decrease much so there is not much to gain in adding the extra computational complexity. The per-class performance seems to be also stable, although there seems to be an imbalance towards selecting class 0. This may be just a side effect of the prior probability difference $P(1) << P(0)$, given that the number of samples for class 0 is much larger in this dataset.

Table 7.1: Performance for the haberman dataset (bagging - majorityvoting)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	4	0.241	0.059	0.763	0.583	0.529	0.557	0.538
1	train	4	0.269	0.073	0.732	0.536	0.516	0.578	0.541
2	test	3	0.352	0.149	0.737	0.546	0.685	0.498	0.566
2	train	3	0.312	0.098	0.734	0.539	0.657	0.514	0.577
3	test	2	0.241	0.058	0.776	0.602	0.705	0.580	0.635
3	train	2	0.282	0.080	0.731	0.535	0.669	0.583	0.622
4	test	4	0.257	0.066	0.762	0.580	0.657	0.579	0.615
4	train	4	0.286	0.082	0.749	0.561	0.714	0.565	0.631
5	test	2	0.236	0.056	0.764	0.584	0.645	0.572	0.606
5	train	2	0.250	0.063	0.752	0.567	0.733	0.590	0.652
6	test	3	0.236	0.056	0.781	0.611	0.700	0.574	0.630
6	train	3	0.288	0.083	0.741	0.549	0.718	0.527	0.607
7	test	4	0.233	0.055	0.772	0.597	0.610	0.578	0.589
7	train	4	0.269	0.073	0.739	0.546	0.610	0.570	0.584
8	test	1	0.208	0.043	0.792	0.628	0.723	0.615	0.665
8	train	1	0.270	0.073	0.740	0.548	0.678	0.601	0.637
9	test	3	0.236	0.056	0.763	0.582	0.643	0.592	0.616
9	train	3	0.253	0.064	0.750	0.563	0.689	0.591	0.636
10	test	2	0.241	0.058	0.764	0.584	0.646	0.559	0.599
10	train	2	0.280	0.079	0.748	0.559	0.723	0.556	0.626
15	test	1	0.226	0.051	0.769	0.591	0.636	0.520	0.572
15	train	1	0.280	0.078	0.720	0.518	0.612	0.511	0.557
20	test	2	0.241	0.058	0.764	0.584	0.627	0.573	0.598
20	train	2	0.255	0.065	0.751	0.565	0.744	0.577	0.649
30	test	1	0.226	0.051	0.778	0.606	0.706	0.581	0.638
30	train	1	0.280	0.078	0.728	0.529	0.655	0.551	0.598
60	test	1	0.236	0.056	0.769	0.591	0.676	0.561	0.613
60	train	1	0.270	0.073	0.728	0.529	0.620	0.540	0.577
100	test	1	0.226	0.051	0.778	0.606	0.706	0.581	0.638
100	train	1	0.275	0.076	0.722	0.522	0.596	0.531	0.561

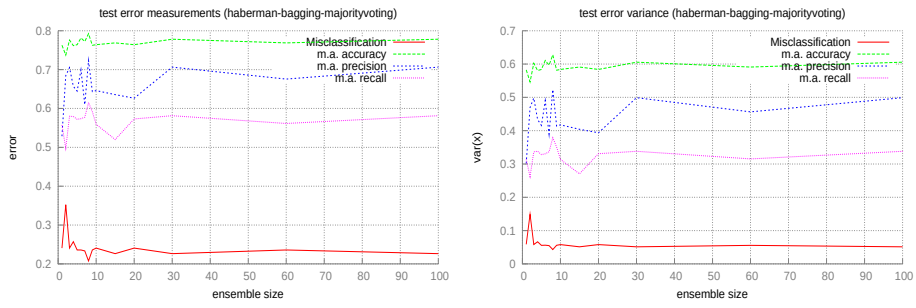


Figure 7.1: Performance on the Haberman dataset (test set) for Bagging with Majority Voting

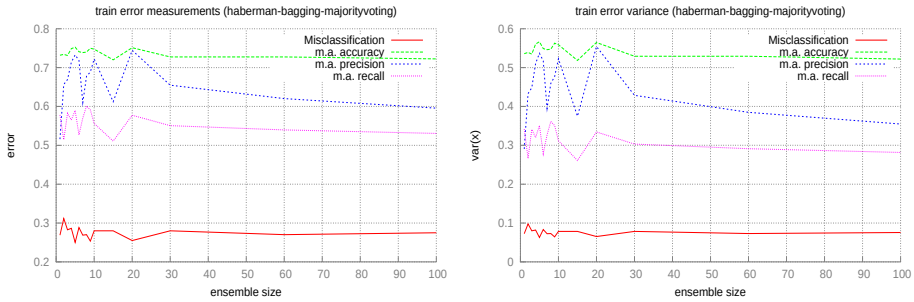


Figure 7.2: Performance on the Haberman dataset (training set) for Bagging with Majority Voting

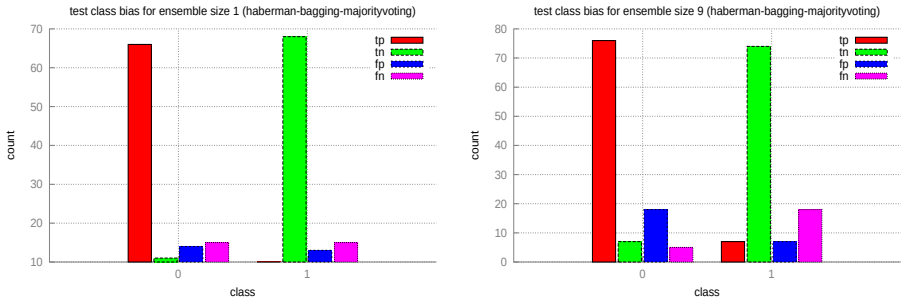


Figure 7.3: Class bias on the Haberman dataset (test set) for Bagging with Majority Voting

Letter Recognition

Letter recognition has many different classes and although the prior probabilities for each class are not identical, they are all in a similar range. This means that it is less likely for any classifier to bias towards a constant single-class answer. The class bias histograms are slightly harder to read because of the high number of true negatives for each class, but it is clear by closer observation that this remains true across different ensemble sizes. Accuracy increases almost monotonically, but recall seems to reach an optimum between an ensemble size of 20 and 30, which also coincides with a local minima in misclassification. [2]

[2]This optimal region could have some correlation with the fact that there are 26 classes, but a method for testing and proving this is not considered in the scope of this project.

Table 7.2: Performance for the letterrecognition dataset (bagging - majorityvoting)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	3	0.488	0.243	0.979	0.958	0.830	0.530	0.646
1	train	3	0.468	0.225	0.980	0.960	0.857	0.547	0.666
2	test	3	0.532	0.289	0.979	0.958	0.913	0.473	0.621
2	train	3	0.512	0.269	0.979	0.959	0.916	0.490	0.636
3	test	2	0.511	0.267	0.979	0.958	0.878	0.504	0.637
3	train	2	0.490	0.246	0.980	0.960	0.902	0.522	0.658
4	test	3	0.504	0.261	0.980	0.960	0.911	0.505	0.646
4	train	3	0.484	0.241	0.980	0.961	0.923	0.520	0.662
5	test	2	0.502	0.258	0.979	0.959	0.883	0.510	0.644
5	train	2	0.482	0.239	0.980	0.961	0.923	0.528	0.668
6	test	2	0.517	0.274	0.979	0.959	0.871	0.493	0.628
6	train	2	0.497	0.254	0.980	0.960	0.911	0.507	0.648
7	test	3	0.472	0.228	0.981	0.961	0.901	0.541	0.674
7	train	3	0.454	0.212	0.981	0.963	0.923	0.555	0.690
8	test	3	0.484	0.242	0.980	0.961	0.906	0.526	0.663
8	train	3	0.465	0.224	0.981	0.963	0.908	0.540	0.675
9	test	2	0.501	0.257	0.980	0.959	0.916	0.511	0.652
9	train	2	0.479	0.236	0.980	0.961	0.926	0.529	0.670
10	test	2	0.522	0.279	0.979	0.958	0.900	0.488	0.629
10	train	2	0.499	0.256	0.980	0.960	0.914	0.506	0.648
15	test	1	0.592	0.350	0.976	0.953	0.912	0.416	0.571
15	train	1	0.570	0.325	0.977	0.955	0.930	0.434	0.592
20	test	2	0.506	0.263	0.979	0.959	0.895	0.504	0.642
20	train	2	0.487	0.244	0.980	0.961	0.919	0.518	0.659
30	test	2	0.507	0.265	0.979	0.959	0.901	0.503	0.642
30	train	2	0.486	0.243	0.980	0.961	0.939	0.519	0.664
60	test	1	0.591	0.350	0.976	0.953	0.920	0.416	0.573
60	train	1	0.568	0.323	0.977	0.955	0.939	0.435	0.595
100	test	1	0.587	0.345	0.977	0.954	0.868	0.420	0.566
100	train	1	0.566	0.320	0.977	0.955	0.937	0.437	0.596

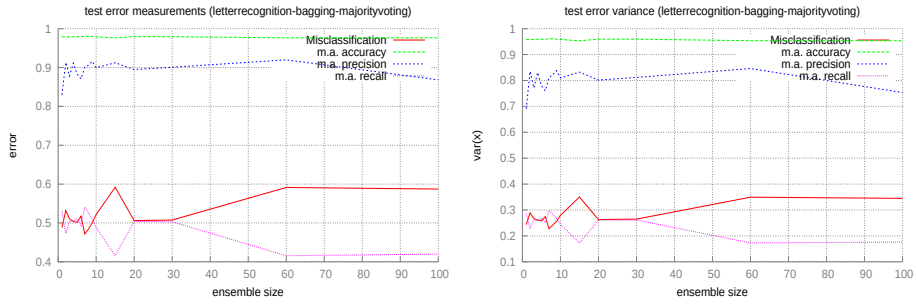


Figure 7.4: Performance on the Letter Recognition dataset (test set) for Bagging with Majority Voting

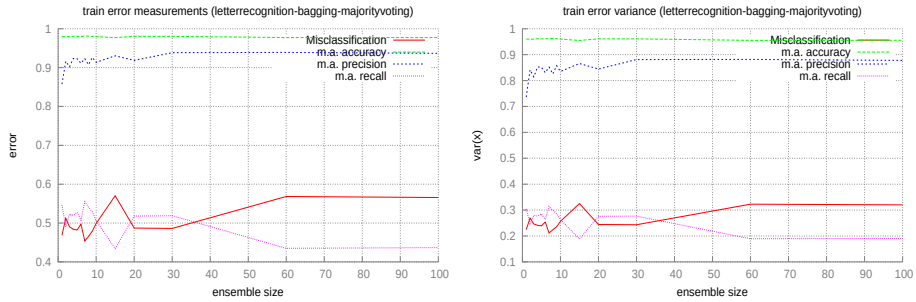


Figure 7.5: Performance on the Letter Recognition dataset (training set) for Bagging with Majority Voting

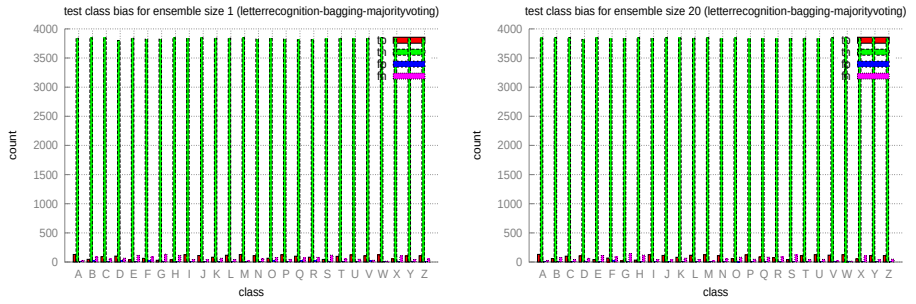


Figure 7.6: Class bias on the Letter Recognition dataset (test set) for Bagging with Majority Voting

Landsat

Performance on this dataset is very poor, and this may be due to the absolute poor performance of the single classifier, although it is not completely understood at the time of writing. We seem to reach an optimal point at ensemble size 30, but the overall result is far from satisfactory. Further investigation would be required as to why this data is not being fit properly. This dataset has fewer classes than the previous, so it is easier to spot class bias. It may be tricky though because some classes don't have many example (class 5 has none). However, we can see that classes 3 and 4 are being very poorly fit (and there is a bias to ignore them). This is indicated by the fact that the number of false negatives is high and the number of true positives is 0 or close to 0.

Table 7.3: Performance for the landsat dataset (bagging - majorityvoting)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	4	0.447	0.223	0.928	0.861	0.588	0.431	0.496
1	train	4	0.407	0.199	0.933	0.872	0.607	0.450	0.514
2	test	4	0.434	0.201	0.932	0.868	0.599	0.439	0.507
2	train	4	0.394	0.172	0.937	0.878	0.603	0.462	0.523
3	test	3	0.420	0.184	0.932	0.869	0.612	0.449	0.518
3	train	3	0.376	0.150	0.938	0.881	0.622	0.470	0.535
4	test	4	0.466	0.239	0.928	0.862	0.637	0.425	0.507
4	train	4	0.409	0.186	0.935	0.875	0.612	0.452	0.519
5	test	3	0.459	0.220	0.928	0.861	0.571	0.428	0.489
5	train	3	0.403	0.172	0.935	0.875	0.577	0.454	0.507
6	test	4	0.438	0.209	0.931	0.867	0.630	0.442	0.517
6	train	4	0.384	0.166	0.938	0.880	0.633	0.470	0.537
7	test	3	0.439	0.206	0.929	0.863	0.610	0.442	0.512
7	train	3	0.377	0.153	0.938	0.879	0.593	0.470	0.524
8	test	4	0.425	0.191	0.932	0.868	0.589	0.451	0.511
8	train	4	0.369	0.144	0.939	0.882	0.599	0.476	0.530
9	test	3	0.421	0.185	0.933	0.870	0.615	0.452	0.521
9	train	3	0.363	0.139	0.940	0.884	0.619	0.479	0.540
10	test	4	0.452	0.227	0.928	0.862	0.592	0.439	0.504
10	train	4	0.386	0.165	0.937	0.878	0.636	0.466	0.535
15	test	1	0.521	0.272	0.922	0.849	0.532	0.381	0.444
15	train	1	0.465	0.217	0.929	0.863	0.538	0.397	0.457
20	test	2	0.513	0.289	0.921	0.848	0.583	0.398	0.471
20	train	2	0.441	0.214	0.930	0.866	0.562	0.424	0.483
30	test	3	0.438	0.203	0.931	0.866	0.615	0.445	0.516
30	train	3	0.376	0.151	0.939	0.881	0.622	0.470	0.535
60	test	3	0.435	0.200	0.931	0.867	0.616	0.446	0.516
60	train	3	0.377	0.152	0.938	0.881	0.618	0.471	0.534
100	test	3	0.440	0.208	0.930	0.865	0.615	0.443	0.514
100	train	3	0.378	0.154	0.938	0.880	0.618	0.470	0.533

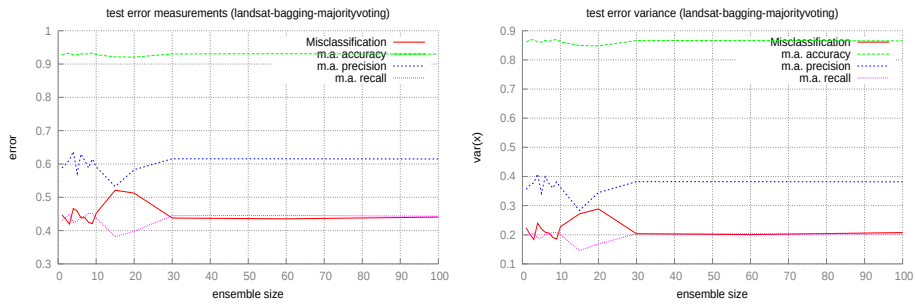


Figure 7.7: Performance on the Landsat dataset (test set) for Bagging with Majority Voting

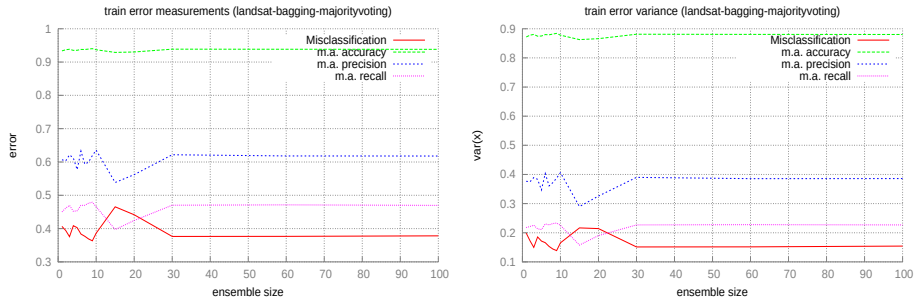


Figure 7.8: Performance on the Landsat dataset (training set) for Bagging with Majority Voting

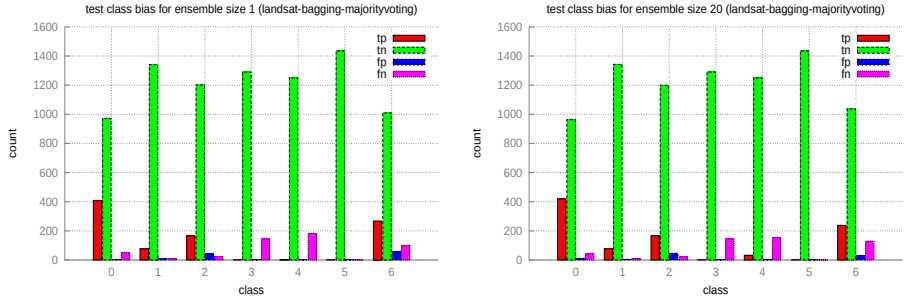


Figure 7.9: Class bias on the Landsat dataset (test set) for Bagging with Majority Voting

Magic

A very disappointing run can be found in this combination of dataset and technique. On the test set, all ensemble sizes, there is a clear bias towards class 1 shown in the histograms. This is much better on the training sets, and when the observation is combined with the actual performance measures, it is clearly an indication that this technique is overfitting the training set, and is therefore biased. Something that could help fix this problem could be to change the target training error to a higher value. This was not tried because of a lack of time when this observation was made at the end of the project.

Table 7.4: Performance for the magic dataset (bagging - majorityvoting)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	3	0.506	0.266	0.509	0.267	0.500	0.258	0.339
1	train	3	0.142	0.020	0.860	0.740	0.838	0.737	0.784
2	test	3	0.512	0.263	0.524	0.275	0.500	0.244	0.328
2	train	3	0.156	0.025	0.859	0.739	0.850	0.717	0.778
3	test	3	0.503	0.254	0.506	0.256	0.500	0.252	0.335
3	train	3	0.141	0.020	0.861	0.741	0.845	0.733	0.785
4	test	2	0.513	0.264	0.504	0.255	0.500	0.245	0.328
4	train	2	0.149	0.022	0.858	0.735	0.842	0.721	0.777
5	test	3	0.489	0.241	0.519	0.271	0.500	0.260	0.342
5	train	3	0.139	0.019	0.863	0.744	0.843	0.741	0.788
6	test	2	0.513	0.265	0.503	0.255	0.500	0.244	0.327
6	train	2	0.148	0.022	0.858	0.736	0.843	0.723	0.778
7	test	3	0.498	0.249	0.508	0.260	0.500	0.254	0.336
7	train	3	0.140	0.020	0.862	0.743	0.848	0.734	0.787
8	test	2	0.502	0.253	0.505	0.256	0.500	0.250	0.333
8	train	2	0.146	0.021	0.857	0.734	0.837	0.726	0.778
9	test	3	0.483	0.234	0.523	0.274	0.500	0.262	0.344
9	train	3	0.138	0.019	0.863	0.745	0.845	0.741	0.790
10	test	2	0.518	0.270	0.490	0.241	0.500	0.243	0.327
10	train	2	0.144	0.021	0.858	0.736	0.842	0.724	0.779
15	test	1	0.551	0.303	0.456	0.208	0.500	0.228	0.313
15	train	1	0.158	0.025	0.845	0.713	0.822	0.702	0.757
20	test	2	0.500	0.252	0.505	0.256	0.500	0.250	0.333
20	train	2	0.143	0.021	0.859	0.738	0.842	0.728	0.781
30	test	3	0.490	0.240	0.513	0.264	0.500	0.256	0.339
30	train	3	0.139	0.020	0.862	0.743	0.847	0.736	0.788
60	test	1	0.528	0.279	0.474	0.225	0.500	0.237	0.322
60	train	1	0.156	0.024	0.844	0.713	0.814	0.708	0.757
100	test	2	0.500	0.251	0.503	0.254	0.500	0.251	0.334
100	train	2	0.142	0.020	0.859	0.737	0.840	0.730	0.781

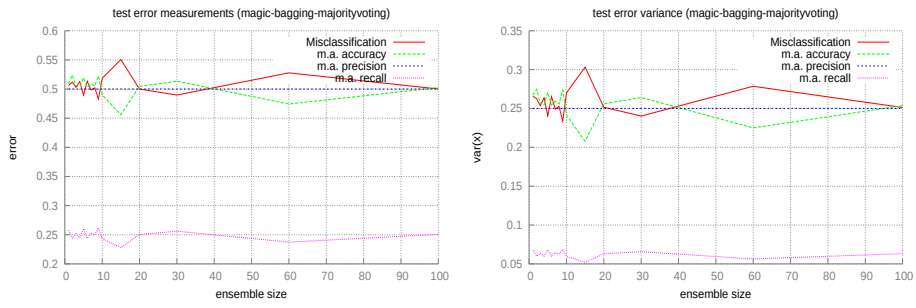


Figure 7.10: Performance on the Magic dataset (test set) for Bagging with Majority Voting

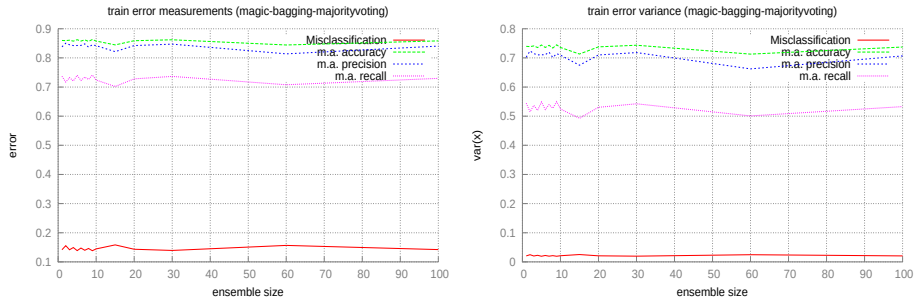


Figure 7.11: Performance on the Magic dataset (training set) for Bagging with Majority Voting

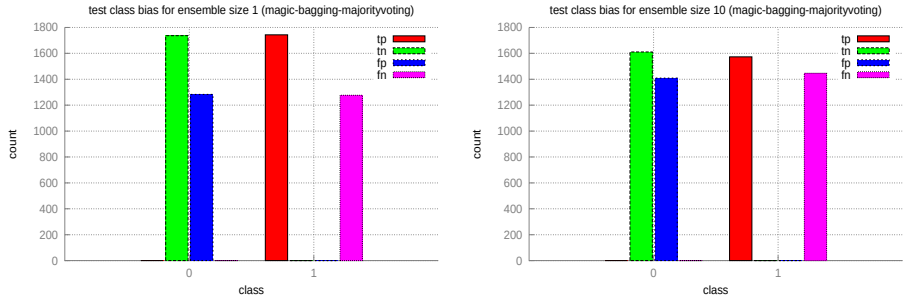


Figure 7.12: Class bias on the Magic dataset (test set) for Bagging with Majority Voting

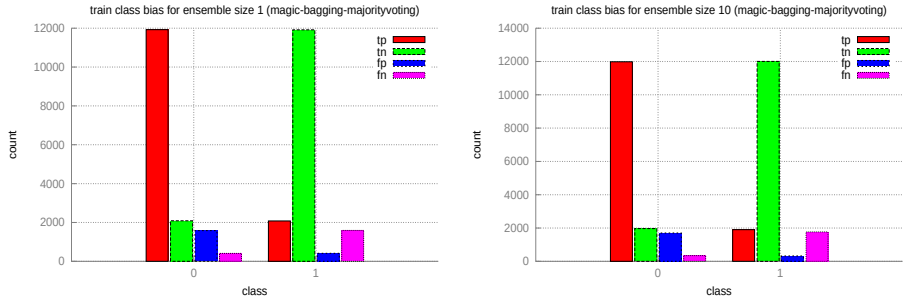


Figure 7.13: Class bias on the Magic dataset (training set) for Bagging with Majority Voting

7.1.2 Averaging

Given the similarity between the two approaches, there are not many differences to be expected in the results.

Haberman

As expected, haberman is classified in a very similar way.

Table 7.5: Performance for the haberman dataset (bagging - averaging)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	6	0.217	0.047	0.782	0.612	0.655	0.616	0.631
1	train	6	0.259	0.067	0.740	0.547	0.648	0.574	0.603
2	test	5	0.230	0.053	0.768	0.590	0.652	0.563	0.604
2	train	5	0.279	0.078	0.724	0.525	0.630	0.531	0.573
3	test	4	0.224	0.050	0.776	0.602	0.677	0.587	0.629
3	train	4	0.254	0.065	0.748	0.559	0.707	0.580	0.636
4	test	4	0.229	0.052	0.772	0.597	0.675	0.615	0.644
4	train	4	0.235	0.055	0.763	0.582	0.721	0.622	0.668
5	test	3	0.211	0.045	0.786	0.618	0.713	0.593	0.647
5	train	3	0.275	0.076	0.732	0.535	0.804	0.530	0.634
6	test	3	0.226	0.052	0.777	0.603	0.641	0.581	0.610
6	train	3	0.252	0.064	0.747	0.558	0.728	0.576	0.642
7	test	3	0.217	0.047	0.780	0.608	0.699	0.609	0.651
7	train	3	0.247	0.061	0.750	0.563	0.702	0.582	0.636
8	test	3	0.226	0.051	0.770	0.594	0.674	0.574	0.618
8	train	3	0.257	0.066	0.746	0.557	0.597	0.569	0.576
9	test	1	0.208	0.043	0.788	0.621	0.716	0.621	0.665
9	train	1	0.275	0.076	0.725	0.526	0.617	0.525	0.568
10	test	2	0.231	0.053	0.759	0.577	0.604	0.566	0.584
10	train	2	0.255	0.065	0.746	0.557	0.669	0.568	0.614
15	test	1	0.198	0.039	0.792	0.628	0.720	0.621	0.667
15	train	1	0.265	0.070	0.730	0.533	0.658	0.547	0.597
20	test	1	0.217	0.047	0.788	0.621	0.697	0.615	0.654
20	train	1	0.240	0.058	0.758	0.574	0.724	0.610	0.662
30	test	3	0.217	0.047	0.785	0.616	0.705	0.622	0.661
30	train	3	0.247	0.061	0.755	0.570	0.744	0.590	0.658
60	test	1	0.208	0.043	0.797	0.635	0.724	0.621	0.669
60	train	1	0.270	0.073	0.735	0.540	0.691	0.542	0.607
100	test	1	0.226	0.051	0.778	0.606	0.706	0.581	0.638
100	train	1	0.270	0.073	0.725	0.526	0.617	0.534	0.573

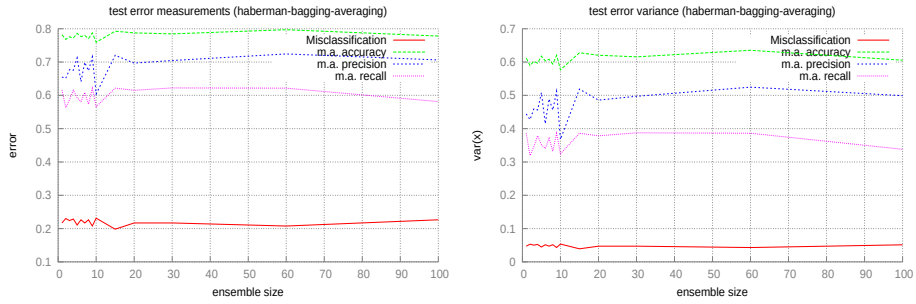


Figure 7.14: Performance on the Haberman dataset (test set) for Bagging with Averaging

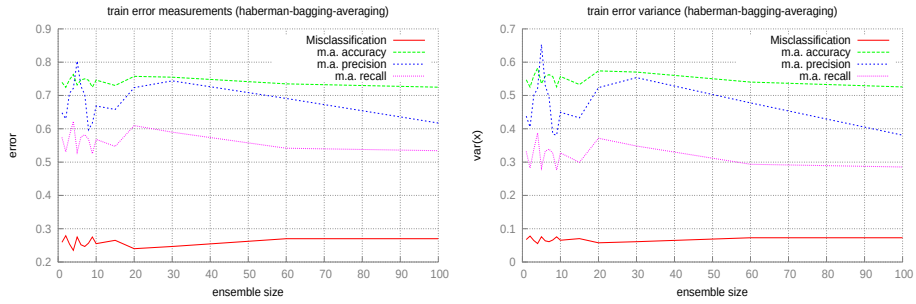


Figure 7.15: Performance on the Haberman dataset (training set) for Bagging with Averaging

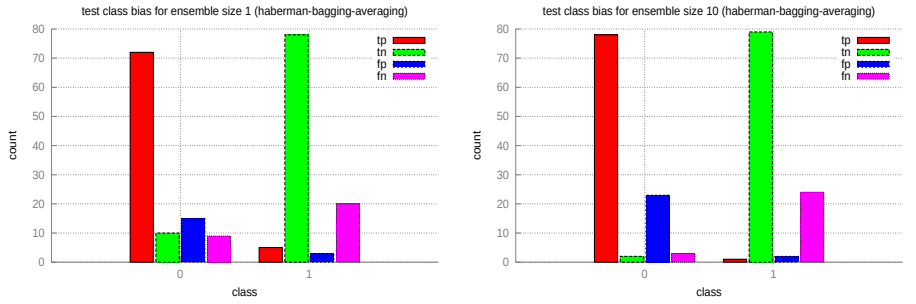


Figure 7.16: Class bias on the Haberman dataset (test set) for Bagging with Averaging

Letter Recognition

This approach yields slightly worse results, but not in a significant way. We see an optimal region at about 7-8 members.

Table 7.6: Performance for the letterrecognition dataset (bagging - averaging)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	3	0.367	0.138	0.979	0.958	0.836	0.539	0.653
1	train	3	0.344	0.121	0.980	0.960	0.854	0.555	0.670
2	test	3	0.396	0.160	0.977	0.955	0.848	0.481	0.612
2	train	3	0.378	0.146	0.978	0.957	0.853	0.494	0.624
3	test	2	0.375	0.144	0.979	0.959	0.892	0.508	0.644
3	train	2	0.354	0.129	0.980	0.961	0.912	0.525	0.663
4	test	2	0.363	0.136	0.979	0.959	0.909	0.500	0.641
4	train	2	0.345	0.123	0.980	0.960	0.922	0.514	0.655
5	test	1	0.429	0.184	0.976	0.953	0.914	0.409	0.565
5	train	1	0.411	0.169	0.977	0.954	0.935	0.423	0.582
6	test	2	0.365	0.137	0.979	0.959	0.895	0.501	0.639
6	train	2	0.346	0.124	0.980	0.961	0.932	0.520	0.663
7	test	3	0.340	0.118	0.981	0.962	0.929	0.540	0.680
7	train	3	0.321	0.105	0.982	0.964	0.931	0.554	0.691
8	test	2	0.361	0.134	0.979	0.959	0.887	0.501	0.638
8	train	2	0.343	0.122	0.980	0.961	0.931	0.519	0.662
9	test	2	0.361	0.134	0.980	0.959	0.903	0.507	0.646
9	train	2	0.342	0.120	0.981	0.961	0.929	0.526	0.668
10	test	2	0.365	0.137	0.980	0.960	0.915	0.509	0.651
10	train	2	0.345	0.123	0.981	0.962	0.936	0.529	0.672
15	test	1	0.426	0.182	0.977	0.954	0.858	0.418	0.562
15	train	1	0.406	0.165	0.977	0.955	0.861	0.434	0.577
20	test	2	0.362	0.134	0.980	0.960	0.928	0.509	0.654
20	train	2	0.341	0.120	0.981	0.962	0.936	0.524	0.668
30	test	1	0.422	0.178	0.977	0.954	0.897	0.421	0.573
30	train	1	0.405	0.164	0.977	0.955	0.936	0.437	0.596
60	test	1	0.419	0.175	0.977	0.954	0.863	0.421	0.566
60	train	1	0.402	0.162	0.978	0.956	0.941	0.438	0.598

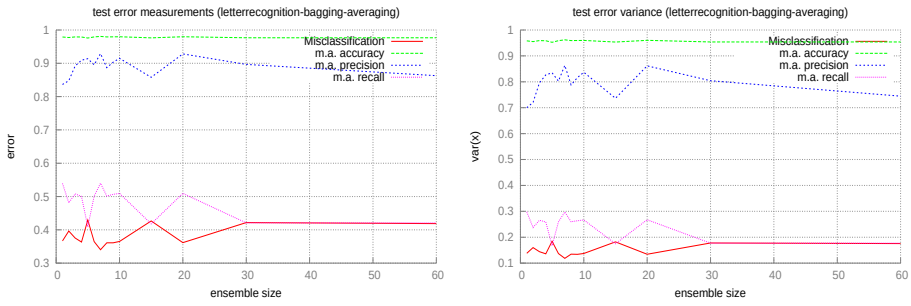


Figure 7.17: Performance on the Letter Recognition dataset (test set) for Bagging with Averaging

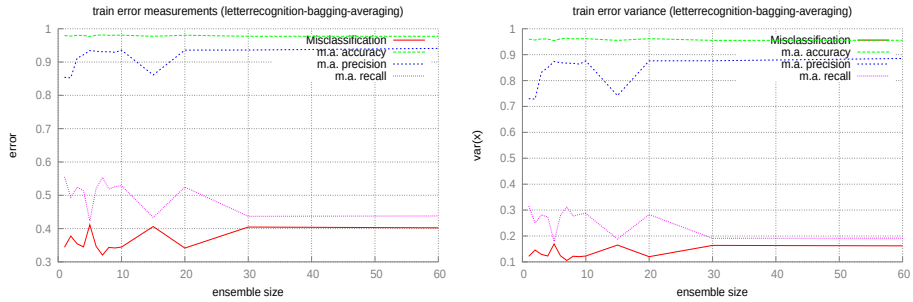


Figure 7.18: Performance on the Letter Recognition dataset (training set) for Bagging with Averaging

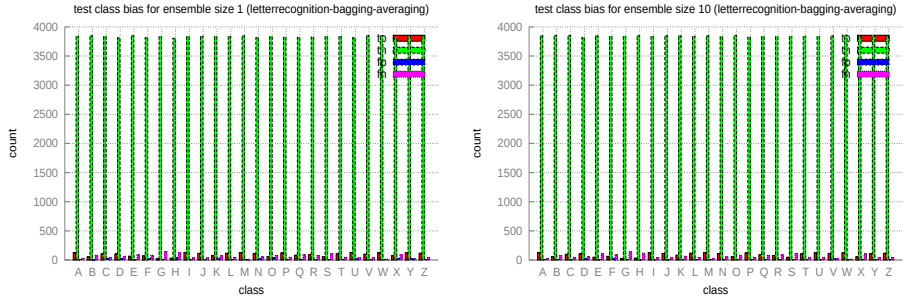


Figure 7.19: Class bias on the Letter Recognition dataset (test set) for Bagging with Averaging

Landsat

For averaging the results have been found to be almost identical to the ones found with majority voting and the illustrations have been omitted.

Magic

The same observations made for majority voting apply here.

Table 7.7: Performance for the magic dataset (bagging - averaging)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	3	0.479	0.230	0.521	0.272	0.500	0.263	0.345
1	train	3	0.139	0.020	0.860	0.740	0.837	0.742	0.786
2	test	3	0.489	0.240	0.511	0.263	0.500	0.253	0.335
2	train	3	0.138	0.019	0.862	0.743	0.848	0.733	0.786
3	test	3	0.475	0.226	0.525	0.275	0.500	0.258	0.340
3	train	3	0.139	0.019	0.861	0.741	0.841	0.735	0.785
4	test	3	0.503	0.254	0.498	0.248	0.500	0.249	0.332
4	train	3	0.139	0.020	0.861	0.741	0.848	0.732	0.786
5	test	3	0.465	0.217	0.535	0.287	0.500	0.270	0.351
5	train	3	0.137	0.019	0.862	0.744	0.837	0.748	0.790
6	test	3	0.493	0.244	0.507	0.257	0.500	0.253	0.336
6	train	3	0.140	0.020	0.860	0.740	0.844	0.733	0.785
7	test	3	0.480	0.232	0.520	0.272	0.500	0.258	0.340
7	train	3	0.137	0.019	0.863	0.745	0.848	0.738	0.789
8	test	3	0.475	0.226	0.524	0.275	0.500	0.262	0.343
8	train	3	0.138	0.019	0.862	0.743	0.842	0.741	0.788
9	test	3	0.489	0.240	0.511	0.262	0.500	0.257	0.339
9	train	3	0.139	0.020	0.861	0.741	0.843	0.736	0.786
10	test	3	0.498	0.248	0.503	0.254	0.500	0.252	0.335
10	train	3	0.138	0.019	0.862	0.743	0.849	0.734	0.787
15	test	1	0.518	0.268	0.482	0.232	0.500	0.242	0.326
15	train	1	0.156	0.024	0.844	0.713	0.808	0.714	0.758
20	test	2	0.505	0.257	0.495	0.246	0.500	0.249	0.332
20	train	2	0.142	0.020	0.859	0.737	0.842	0.729	0.782
30	test	3	0.485	0.236	0.515	0.267	0.500	0.259	0.341
30	train	3	0.137	0.019	0.863	0.745	0.848	0.740	0.790
60	test	2	0.495	0.245	0.506	0.256	0.500	0.252	0.335
60	train	2	0.142	0.020	0.858	0.737	0.838	0.731	0.781
100	test	2	0.494	0.245	0.506	0.257	0.500	0.253	0.335
100	train	2	0.142	0.020	0.858	0.737	0.839	0.730	0.781

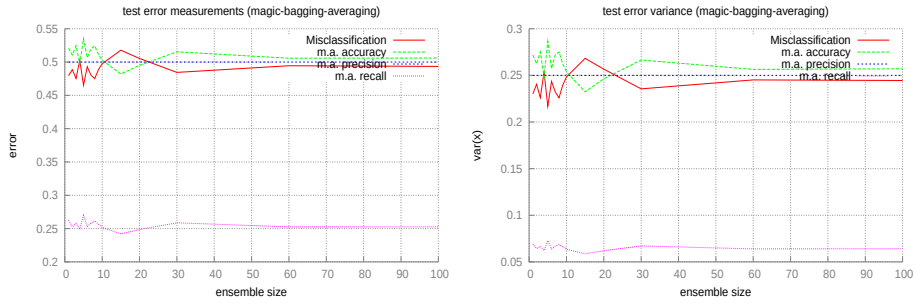


Figure 7.20: Performance on the Magic dataset (test set) for Bagging with Averaging

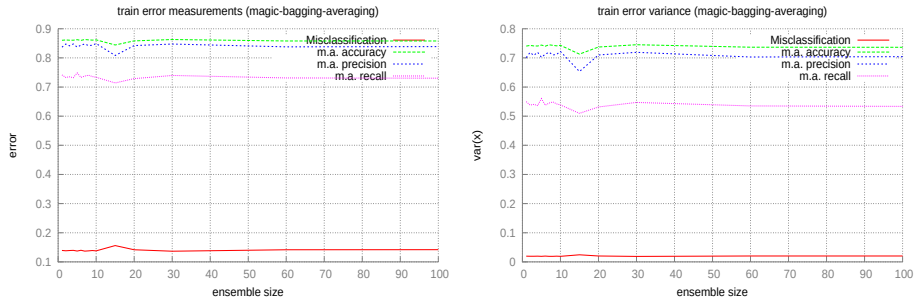


Figure 7.21: Performance on the Magic dataset (training set) for Bagging with Averaging

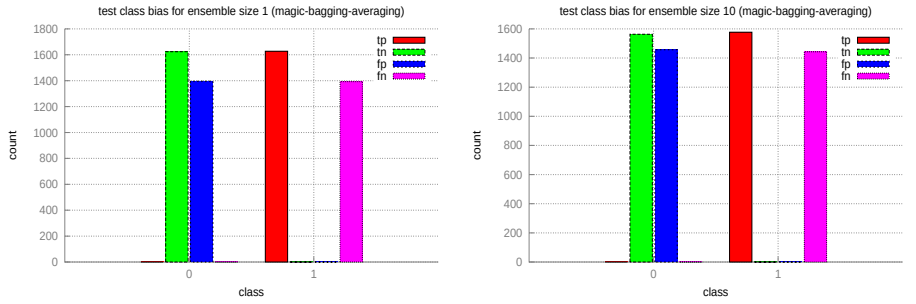


Figure 7.22: Class bias on the Magic dataset (test set) for Bagging with Averaging

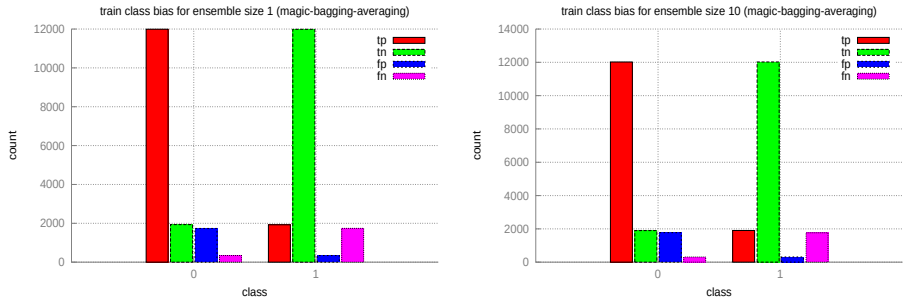


Figure 7.23: Class bias on the Magic dataset (training set) for Bagging with Averaging

7.2 AdaBoost

Compared to the Bagging approach, the hope for AdaBoost is that the ensemble will be able to improve on those examples that are much harder to classify, therefore increasing the overall results.

7.2.1 Majority voting

Haberman

The optimum here is reached at around 30 ensembles, and the class bias noticed in the previous examples is slightly alleviated. This was at partially expected, and shows the characteristic of boosting techniques of being able to adapt and focus on the harder instances.

Table 7.8: Performance for the haberman dataset (adaboost - majorityvoting)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	5	0.228	0.052	0.774	0.599	0.685	0.620	0.650
1	train	5	0.268	0.072	0.737	0.543	0.663	0.629	0.645
2	test	4	0.394	0.183	0.719	0.523	0.547	0.462	0.490
2	train	4	0.311	0.098	0.745	0.555	0.678	0.521	0.580
3	test	3	0.226	0.052	0.772	0.596	0.641	0.593	0.616
3	train	3	0.262	0.069	0.748	0.560	0.713	0.585	0.643
4	test	4	0.269	0.073	0.764	0.584	0.627	0.516	0.554
4	train	4	0.297	0.089	0.739	0.547	0.632	0.521	0.564
5	test	2	0.231	0.053	0.774	0.598	0.685	0.605	0.642
5	train	2	0.247	0.061	0.755	0.570	0.722	0.600	0.656
6	test	3	0.239	0.058	0.772	0.596	0.587	0.572	0.577
6	train	3	0.265	0.070	0.746	0.557	0.606	0.565	0.577
7	test	1	0.226	0.051	0.764	0.584	0.552	0.520	0.536
7	train	1	0.285	0.081	0.715	0.511	0.560	0.511	0.535
8	test	3	0.245	0.060	0.775	0.601	0.657	0.558	0.604
8	train	3	0.285	0.081	0.739	0.547	0.705	0.537	0.609
9	test	1	0.236	0.056	0.774	0.598	0.672	0.581	0.623
9	train	1	0.265	0.070	0.732	0.537	0.661	0.552	0.602
10	test	3	0.236	0.056	0.777	0.603	0.597	0.565	0.576
10	train	3	0.268	0.072	0.748	0.560	0.780	0.548	0.641
15	test	1	0.208	0.043	0.792	0.628	0.720	0.621	0.667
15	train	1	0.265	0.070	0.738	0.544	0.684	0.572	0.623
20	test	1	0.255	0.065	0.759	0.577	0.383	0.488	0.429
20	train	1	0.280	0.078	0.728	0.529	0.617	0.505	0.556
30	test	1	0.208	0.043	0.792	0.628	0.723	0.615	0.665
30	train	1	0.270	0.073	0.742	0.551	0.696	0.545	0.611
60	test	1	0.226	0.051	0.774	0.598	0.672	0.581	0.623
60	train	1	0.270	0.073	0.732	0.537	0.670	0.563	0.612
100	test	1	0.226	0.051	0.783	0.613	0.727	0.601	0.658
100	train	1	0.280	0.078	0.725	0.526	0.653	0.551	0.597

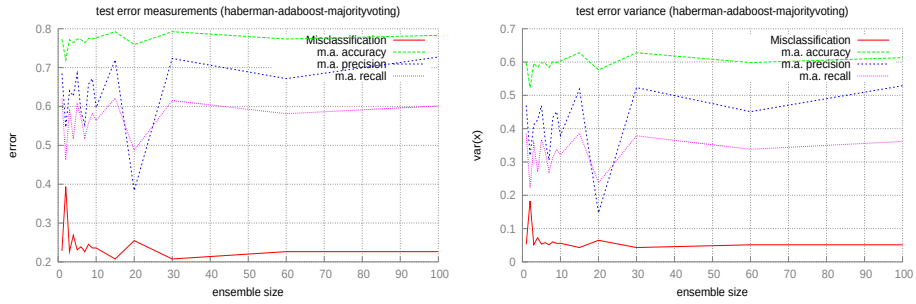


Figure 7.24: Performance on the Haberman dataset (test set) for Bagging with Majority Voting

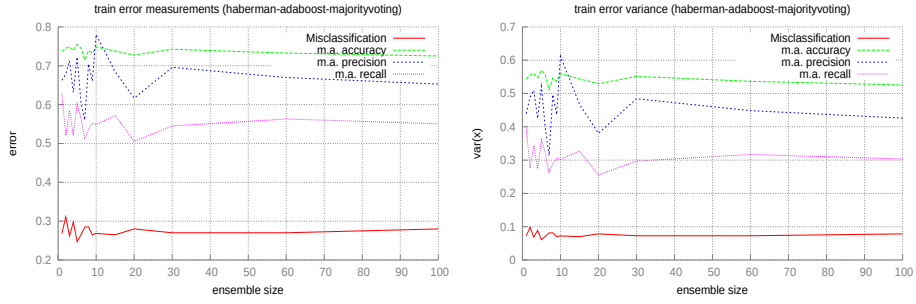


Figure 7.25: Performance on the Haberman dataset (training set) for Bagging with Majority Voting

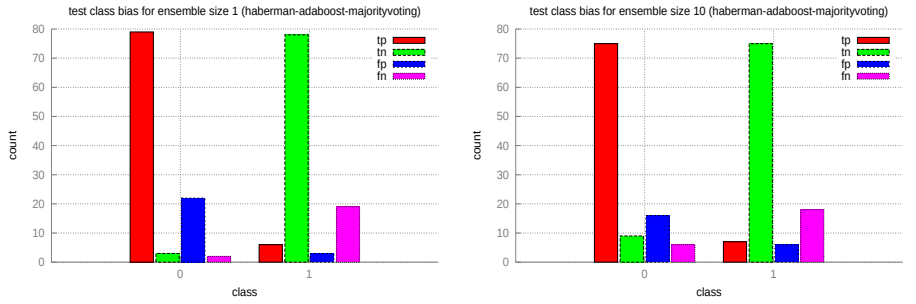


Figure 7.26: Class bias on the Haberman dataset (test set) for Bagging with Majority Voting

Letter Recognition

Very similar observations can be made as the previous examples. It is likely that this is because the letter recognition dataset does not have a large number of

"hard" examples compared to the size of the dataset itself. It is remarkable how in this case the graphs resemble the ones obtained with bagging and majority voting, so they are included for comparison.

Table 7.9: Performance for the letterrecognition dataset (adaboost - majorityvoting)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	4	0.480	0.235	0.979	0.958	0.849	0.550	0.664
1	train	4	0.459	0.216	0.980	0.960	0.868	0.568	0.684
2	test	4	0.519	0.274	0.979	0.959	0.918	0.488	0.635
2	train	4	0.502	0.257	0.980	0.960	0.933	0.502	0.650
3	test	3	0.476	0.233	0.980	0.961	0.891	0.535	0.666
3	train	3	0.457	0.216	0.981	0.962	0.901	0.551	0.681
4	test	3	0.507	0.263	0.979	0.959	0.896	0.502	0.642
4	train	3	0.486	0.242	0.980	0.961	0.917	0.519	0.661
5	test	2	0.504	0.262	0.979	0.959	0.887	0.508	0.643
5	train	2	0.483	0.242	0.980	0.961	0.910	0.524	0.662
6	test	3	0.503	0.260	0.980	0.960	0.886	0.506	0.642
6	train	3	0.482	0.240	0.981	0.962	0.910	0.523	0.661
7	test	2	0.499	0.254	0.980	0.960	0.902	0.511	0.650
7	train	2	0.477	0.234	0.981	0.961	0.924	0.529	0.670
8	test	2	0.513	0.270	0.979	0.959	0.914	0.496	0.639
8	train	2	0.494	0.251	0.980	0.961	0.930	0.510	0.656
9	test	2	0.502	0.258	0.980	0.960	0.885	0.509	0.644
9	train	2	0.480	0.237	0.980	0.961	0.900	0.527	0.662
10	test	3	0.483	0.240	0.980	0.961	0.908	0.528	0.665
10	train	3	0.465	0.223	0.981	0.963	0.929	0.542	0.681
15	test	1	0.581	0.338	0.977	0.954	0.856	0.427	0.569
15	train	1	0.562	0.315	0.977	0.955	0.892	0.443	0.592
20	test	2	0.506	0.263	0.979	0.959	0.897	0.503	0.642
20	train	2	0.485	0.242	0.980	0.961	0.918	0.521	0.662
30	test	2	0.505	0.262	0.979	0.959	0.892	0.504	0.641
30	train	2	0.484	0.241	0.980	0.961	0.918	0.521	0.661
60	test	1	0.586	0.344	0.977	0.954	0.856	0.422	0.565
60	train	1	0.566	0.320	0.977	0.955	0.937	0.438	0.597
100	test	1	0.591	0.350	0.976	0.953	0.854	0.416	0.559
100	train	1	0.569	0.323	0.977	0.955	0.938	0.435	0.594

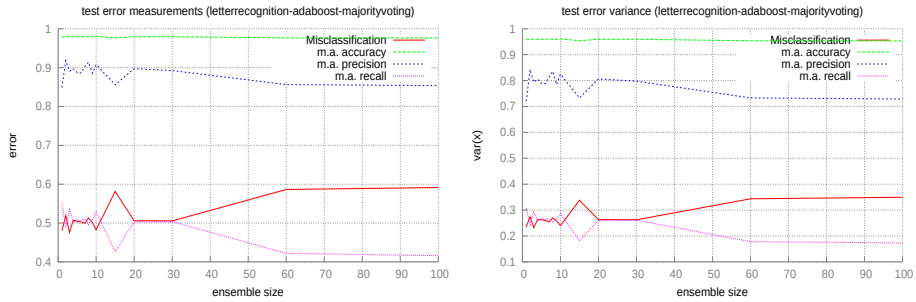


Figure 7.27: Performance on the Letter Recognition dataset (test set) for Bagging with Majority Voting

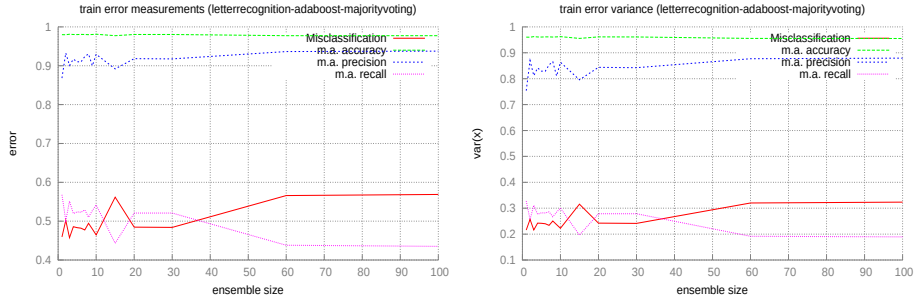


Figure 7.28: Performance on the Letter Recognition dataset (training set) for Bagging with Majority Voting

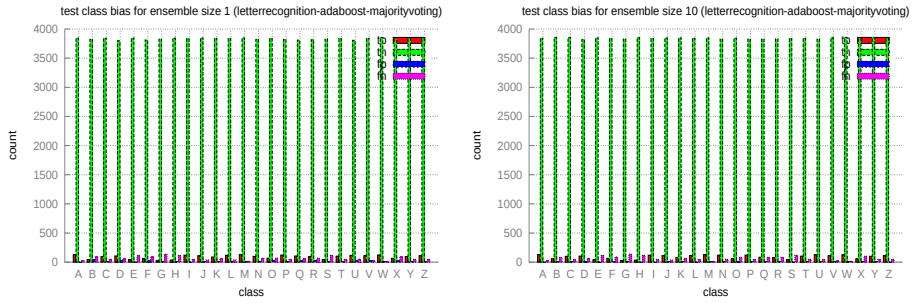


Figure 7.29: Class bias on the Letter Recognition dataset (test set) for Bagging with Majority Voting

Landsat

With fewer than 10 members there there is good performance in general (for this data set), but this seems to get worse with more members. The class bias

found with bagging is slightly alleviated, but not by much. Nevertheless this small improvement shows again that boosting techniques are able to correct errors on difficult data.

Table 7.10: Performance for the landsat dataset (adaboost - majorityvoting)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	3	0.452	0.218	0.925	0.855	0.551	0.435	0.486
1	train	3	0.390	0.164	0.933	0.872	0.565	0.460	0.506
2	test	3	0.482	0.248	0.925	0.856	0.622	0.418	0.499
2	train	3	0.416	0.185	0.934	0.873	0.607	0.444	0.512
3	test	1	0.615	0.379	0.907	0.823	0.526	0.327	0.403
3	train	1	0.547	0.299	0.918	0.842	0.493	0.351	0.410
4	test	3	0.473	0.238	0.926	0.858	0.576	0.420	0.485
4	train	3	0.414	0.188	0.934	0.873	0.579	0.449	0.505
5	test	3	0.447	0.216	0.928	0.861	0.562	0.438	0.492
5	train	3	0.386	0.164	0.936	0.877	0.579	0.465	0.515
6	test	2	0.517	0.288	0.920	0.847	0.527	0.398	0.453
6	train	2	0.441	0.211	0.931	0.866	0.537	0.424	0.474
7	test	2	0.486	0.253	0.925	0.855	0.598	0.409	0.484
7	train	2	0.429	0.199	0.932	0.870	0.548	0.433	0.484
8	test	2	0.494	0.256	0.923	0.853	0.596	0.412	0.487
8	train	2	0.415	0.181	0.934	0.872	0.599	0.441	0.507
9	test	3	0.433	0.200	0.931	0.866	0.616	0.449	0.519
9	train	3	0.371	0.148	0.939	0.881	0.615	0.475	0.536
10	test	2	0.503	0.272	0.923	0.852	0.533	0.396	0.454
10	train	2	0.446	0.217	0.931	0.867	0.542	0.422	0.474
15	test	1	0.595	0.354	0.911	0.830	0.531	0.342	0.416
15	train	1	0.518	0.269	0.921	0.849	0.519	0.364	0.428
20	test	2	0.514	0.287	0.920	0.847	0.594	0.404	0.479
20	train	2	0.433	0.204	0.931	0.868	0.593	0.431	0.498
30	test	1	0.622	0.387	0.908	0.824	0.535	0.328	0.407
30	train	1	0.532	0.283	0.920	0.847	0.538	0.354	0.427
60	test	1	0.589	0.347	0.913	0.834	0.538	0.346	0.421
60	train	1	0.512	0.262	0.923	0.852	0.539	0.367	0.437
100	test	1	0.594	0.353	0.912	0.831	0.535	0.344	0.419
100	train	1	0.511	0.261	0.923	0.852	0.535	0.368	0.436

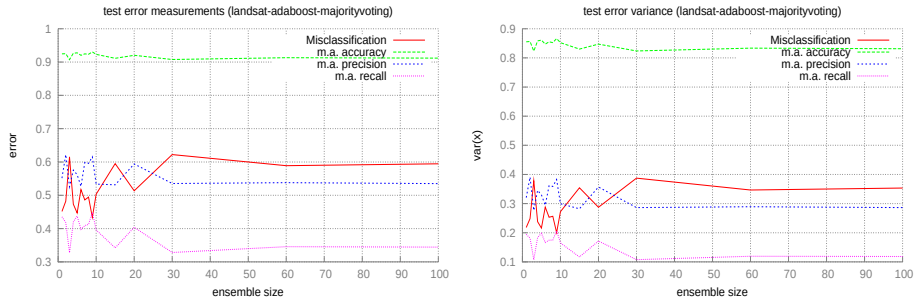


Figure 7.30: Performance on the Landsat dataset (test set) for Bagging with Majority Voting

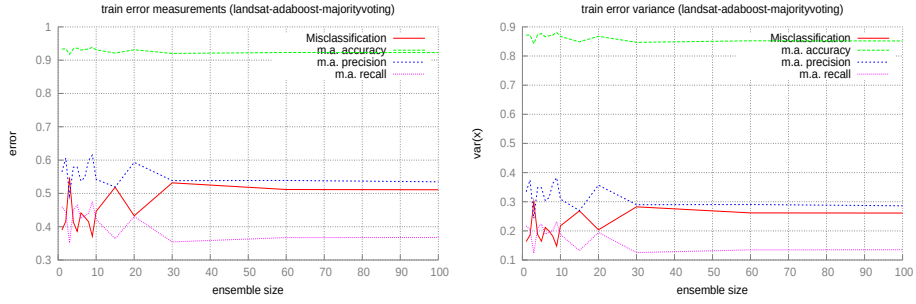


Figure 7.31: Performance on the Landsat dataset (training set) for Bagging with Majority Voting

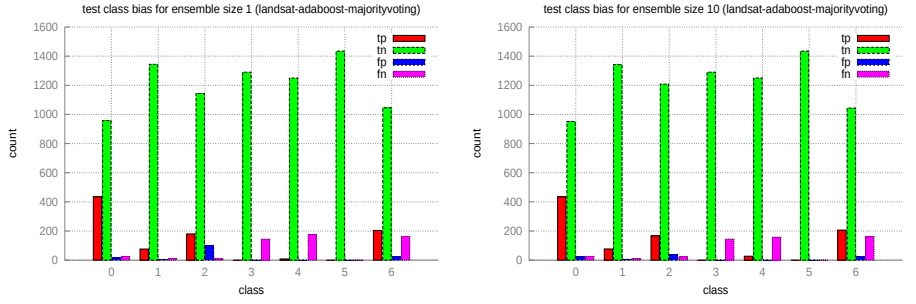


Figure 7.32: Class bias on the Landsat dataset (test set) for Bagging with Majority Voting

Magic

Despite the considerations about overfitting of data on this dataset, boosting techniques are able to recover at least partially. Measurements seem to improve

almost monotonically with ensemble size, and this can be attributed to the combination of increased variance (due to the higher number and complexity of classifiers) and the "specialization" approach of boosting. Class bias diagrams have been omitted for this benchmark as there are no remarkable observations to be made in comparison with prevoius benchmarks on the same set.

Table 7.11: Performance for the magic dataset (adaboost - majorityvoting)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	2	0.493	0.245	0.518	0.270	0.500	0.264	0.345
1	train	2	0.132	0.017	0.870	0.756	0.862	0.750	0.802
2	test	1	0.493	0.243	0.545	0.297	0.500	0.254	0.337
2	train	1	0.143	0.020	0.872	0.760	0.882	0.733	0.801
3	test	2	0.485	0.235	0.524	0.274	0.500	0.261	0.343
3	train	2	0.134	0.018	0.868	0.754	0.858	0.746	0.798
4	test	2	0.492	0.242	0.529	0.279	0.500	0.254	0.337
4	train	2	0.137	0.019	0.871	0.758	0.872	0.738	0.799
5	test	2	0.477	0.227	0.532	0.283	0.500	0.267	0.348
5	train	2	0.131	0.017	0.870	0.757	0.860	0.751	0.802
6	test	2	0.474	0.225	0.540	0.291	0.500	0.264	0.345
6	train	2	0.134	0.018	0.871	0.759	0.870	0.744	0.802
7	test	2	0.472	0.223	0.537	0.288	0.500	0.267	0.349
7	train	2	0.132	0.017	0.871	0.758	0.863	0.750	0.802
8	test	1	0.490	0.240	0.524	0.274	0.500	0.255	0.338
8	train	1	0.134	0.018	0.870	0.757	0.866	0.742	0.799
9	test	1	0.469	0.220	0.536	0.287	0.500	0.268	0.349
9	train	1	0.130	0.017	0.872	0.760	0.863	0.752	0.804
20	test	1	0.474	0.225	0.532	0.283	0.500	0.264	0.346
20	train	1	0.132	0.017	0.870	0.757	0.862	0.748	0.801

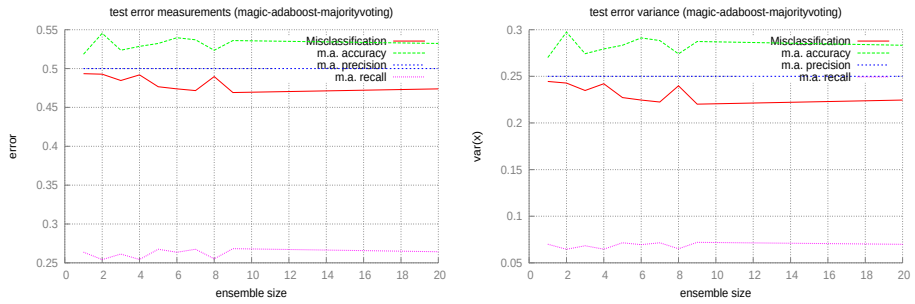


Figure 7.33: Performance on the Magic dataset (test set) for Bagging with Majority Voting

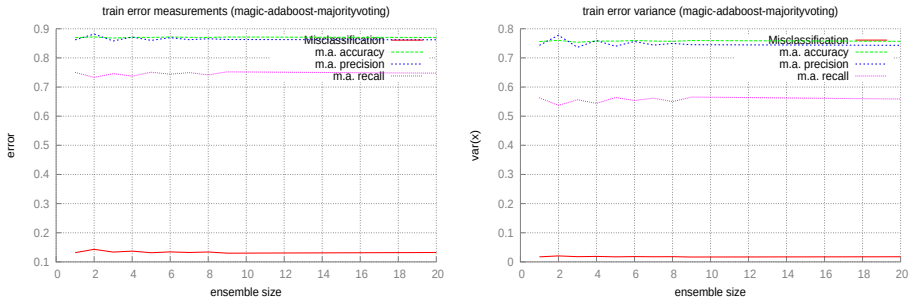


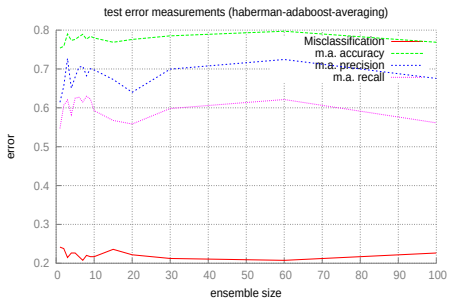
Figure 7.34: Performance on the Magic dataset (training set) for Bagging with Majority Voting

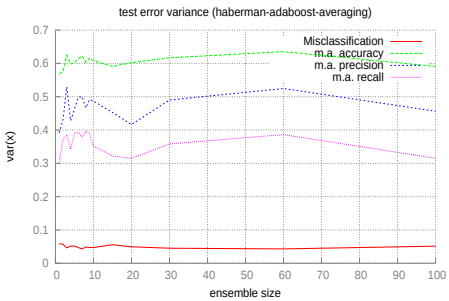
7.2.2 Averaging

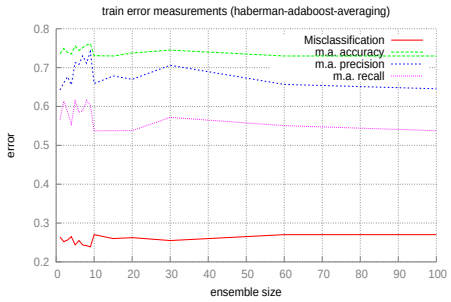
Haberman

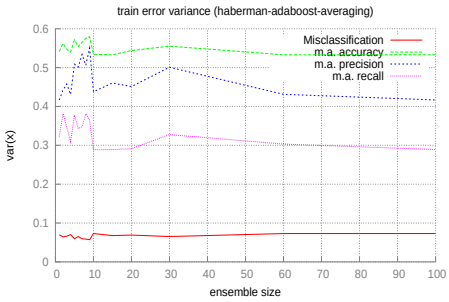
Table 7.12: Performance for the haberman dataset (adaboost - averaging)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	5	0.242	0.058	0.754	0.569	0.614	0.547	0.572
1	train	5	0.264	0.070	0.736	0.541	0.643	0.566	0.601
2	test	5	0.238	0.057	0.758	0.576	0.656	0.607	0.629
2	train	5	0.252	0.064	0.750	0.562	0.661	0.614	0.636
3	test	4	0.215	0.046	0.790	0.624	0.727	0.620	0.669
3	train	4	0.256	0.066	0.739	0.547	0.676	0.586	0.628
4	test	3	0.226	0.052	0.774	0.599	0.650	0.582	0.614
4	train	3	0.265	0.070	0.735	0.540	0.656	0.555	0.601
5	test	4	0.226	0.051	0.776	0.602	0.679	0.625	0.651
5	train	4	0.244	0.060	0.756	0.571	0.713	0.614	0.660
6	test	3	0.217	0.047	0.783	0.613	0.705	0.627	0.663
6	train	3	0.255	0.065	0.744	0.554	0.708	0.585	0.640
7	test	3	0.208	0.043	0.789	0.623	0.707	0.615	0.658
7	train	3	0.243	0.059	0.752	0.565	0.730	0.589	0.652
8	test	3	0.220	0.049	0.777	0.603	0.682	0.630	0.655
8	train	3	0.242	0.059	0.758	0.575	0.711	0.616	0.660
9	test	4	0.217	0.047	0.784	0.615	0.701	0.623	0.659
9	train	4	0.239	0.057	0.761	0.580	0.742	0.603	0.665
10	test	3	0.217	0.047	0.780	0.608	0.698	0.593	0.641
10	train	3	0.270	0.073	0.731	0.534	0.659	0.537	0.591
15	test	1	0.236	0.056	0.769	0.591	0.673	0.568	0.616
15	train	1	0.260	0.068	0.730	0.533	0.678	0.538	0.600
20	test	2	0.222	0.049	0.776	0.602	0.640	0.558	0.596
20	train	2	0.263	0.069	0.738	0.544	0.670	0.538	0.597
30	test	2	0.212	0.045	0.785	0.617	0.700	0.598	0.645
30	train	2	0.255	0.065	0.745	0.555	0.706	0.572	0.632
60	test	1	0.208	0.043	0.797	0.635	0.724	0.621	0.669
60	train	1	0.270	0.073	0.730	0.533	0.657	0.551	0.599
100	test	1	0.226	0.051	0.769	0.591	0.676	0.561	0.613
100	train	1	0.270	0.073	0.730	0.533	0.646	0.538	0.587





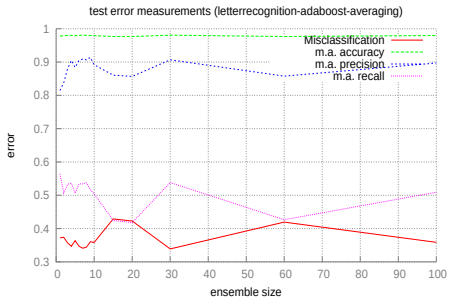


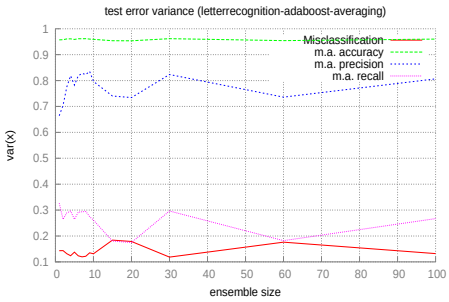


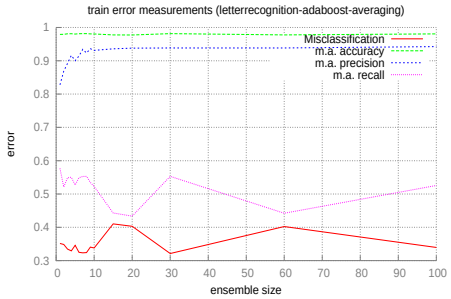
Letter Recognition

Table 7.13: Performance for the letterrecognition dataset (adaboost - averaging)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	4	0.372	0.143	0.978	0.957	0.815	0.565	0.664
1	train	4	0.351	0.127	0.979	0.959	0.828	0.577	0.677
2	test	2	0.374	0.144	0.979	0.958	0.838	0.507	0.630
2	train	2	0.348	0.126	0.980	0.960	0.869	0.521	0.648
3	test	3	0.357	0.132	0.980	0.961	0.879	0.532	0.661
3	train	3	0.335	0.116	0.981	0.963	0.892	0.549	0.678
4	test	3	0.347	0.124	0.980	0.961	0.905	0.537	0.670
4	train	3	0.329	0.112	0.981	0.963	0.916	0.550	0.684
5	test	2	0.364	0.137	0.979	0.959	0.884	0.506	0.641
5	train	2	0.346	0.124	0.980	0.961	0.900	0.527	0.661
6	test	3	0.347	0.123	0.981	0.961	0.904	0.534	0.669
6	train	3	0.325	0.108	0.981	0.963	0.914	0.549	0.684
7	test	3	0.341	0.119	0.981	0.962	0.909	0.534	0.670
7	train	3	0.323	0.108	0.982	0.963	0.934	0.553	0.690
8	test	3	0.344	0.122	0.981	0.962	0.907	0.537	0.672
8	train	3	0.324	0.108	0.982	0.963	0.923	0.553	0.689
9	test	2	0.360	0.135	0.980	0.960	0.914	0.517	0.656
9	train	2	0.341	0.121	0.981	0.962	0.936	0.535	0.676
10	test	2	0.358	0.131	0.979	0.959	0.892	0.504	0.642
10	train	2	0.338	0.117	0.980	0.961	0.931	0.523	0.665
15	test	1	0.429	0.184	0.977	0.954	0.860	0.425	0.569
15	train	1	0.410	0.168	0.978	0.956	0.936	0.443	0.601
20	test	1	0.423	0.179	0.977	0.954	0.857	0.418	0.562
20	train	1	0.403	0.162	0.977	0.955	0.938	0.434	0.593
30	test	3	0.339	0.118	0.981	0.962	0.907	0.538	0.673
30	train	3	0.321	0.107	0.982	0.964	0.938	0.553	0.692
60	test	1	0.419	0.176	0.977	0.954	0.858	0.427	0.570
60	train	1	0.402	0.162	0.978	0.956	0.938	0.442	0.601
100	test	2	0.359	0.132	0.980	0.960	0.897	0.509	0.647
100	train	2	0.339	0.119	0.981	0.962	0.942	0.526	0.671


test error measurements (letterrecognition-adaboost-averaging)
Misclassification
m.a. accuracy
m.a. precision
m.a. recall
error
ensemble size


test error variance (letterrecognition-adaboost-averaging)
Misclassification
m.a. accuracy
m.a. precision
m.a. recall
var(x)
ensemble size

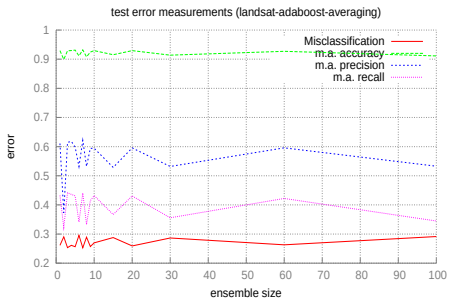

train error measurements (letterrecognition-adaboost-averaging)
Misclassification
m.a. accuracy
m.a. precision
m.a. recall
error
ensemble size

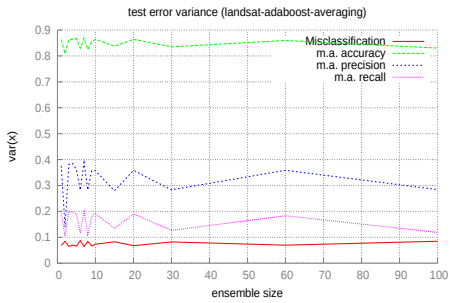

train error variance (letterrecognition-adaboost-averaging)
Misclassification
m.a. accuracy
m.a. precision
m.a. recall
var(x)
ensemble size

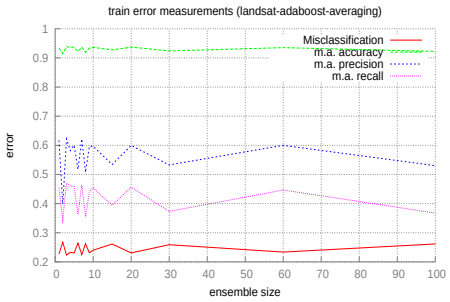
Landsat

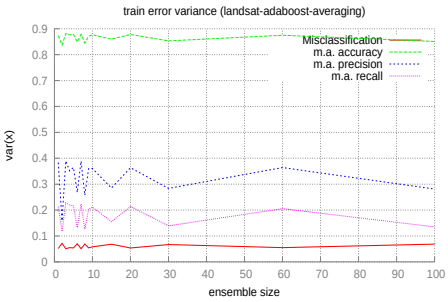
Table 7.14: Performance for the landsat dataset (adaboost - averaging)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	3	0.261	0.068	0.929	0.864	0.611	0.432	0.505
1	train	3	0.227	0.052	0.935	0.874	0.617	0.455	0.523
2	test	1	0.291	0.084	0.899	0.809	0.373	0.323	0.346
2	train	1	0.267	0.072	0.914	0.836	0.398	0.344	0.369
3	test	3	0.253	0.065	0.929	0.863	0.614	0.442	0.513
3	train	3	0.224	0.051	0.939	0.881	0.622	0.472	0.536
4	test	3	0.261	0.069	0.930	0.865	0.618	0.436	0.510
4	train	3	0.233	0.055	0.936	0.877	0.582	0.460	0.514
5	test	2	0.256	0.066	0.931	0.867	0.598	0.431	0.501
5	train	2	0.231	0.054	0.937	0.878	0.600	0.459	0.520
6	test	1	0.295	0.087	0.911	0.830	0.532	0.341	0.416
6	train	1	0.264	0.070	0.921	0.849	0.518	0.363	0.427
7	test	3	0.253	0.064	0.932	0.869	0.622	0.441	0.515
7	train	3	0.224	0.051	0.937	0.878	0.622	0.464	0.530
8	test	1	0.289	0.084	0.908	0.824	0.532	0.333	0.409
8	train	1	0.262	0.069	0.919	0.845	0.507	0.355	0.418
9	test	2	0.257	0.067	0.924	0.855	0.594	0.414	0.486
9	train	2	0.231	0.054	0.933	0.872	0.596	0.441	0.506
10	test	2	0.270	0.073	0.929	0.864	0.594	0.432	0.500
10	train	2	0.240	0.058	0.936	0.877	0.597	0.454	0.515
15	test	1	0.288	0.083	0.915	0.838	0.529	0.367	0.433
15	train	1	0.261	0.068	0.928	0.860	0.534	0.394	0.454
20	test	2	0.259	0.068	0.929	0.864	0.596	0.430	0.499
20	train	2	0.231	0.054	0.937	0.878	0.600	0.456	0.518
30	test	1	0.286	0.082	0.914	0.835	0.532	0.356	0.426
30	train	1	0.259	0.067	0.924	0.853	0.533	0.373	0.439
60	test	2	0.263	0.070	0.927	0.860	0.596	0.422	0.493
60	train	2	0.234	0.055	0.935	0.875	0.600	0.447	0.512
100	test	1	0.291	0.085	0.911	0.830	0.533	0.345	0.419
100	train	1	0.262	0.068	0.922	0.851	0.530	0.368	0.434






train error measurements (landsat-adaboost-averaging)
Misclassification
m.a. accuracy
m.a. precision
m.a. recall
error
ensemble size
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
10
20
30
40
50
60
70
80
90
100


train error variance (landsat-adaboost-averaging)
Misclassification
m.a. accuracy
m.a. precision
m.a. recall
var(x)
ensemble size
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
10
20
30
40
50
60
70
80
90
100

Magic

7.3 Stacking

7.3.1 MetaClassifier

Haberman

Table 7.15: Performance for the landsat dataset (stacking - metaclassifier-mlp30-0.07)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	1	0.606	0.368	0.883	0.780	0.327	0.313	0.320
1	train	1	0.462	0.214	0.904	0.817	0.350	0.318	0.333
2	test	1	0.810	0.656	0.860	0.740	0.146	0.266	0.189
2	train	1	0.648	0.420	0.887	0.787	0.203	0.273	0.233
3	test	1	0.813	0.660	0.791	0.625	0.128	0.271	0.174
3	train	1	0.651	0.423	0.835	0.696	0.176	0.275	0.214
4	test	1	0.814	0.662	0.795	0.631	0.156	0.264	0.196
4	train	1	0.655	0.429	0.835	0.698	0.185	0.268	0.219
5	test	1	0.814	0.662	0.769	0.591	0.151	0.264	0.192
5	train	1	0.656	0.431	0.813	0.662	0.177	0.269	0.214
6	test	1	0.819	0.670	0.772	0.595	0.164	0.243	0.196
6	train	1	0.664	0.441	0.814	0.663	0.182	0.250	0.210
7	test	1	0.816	0.666	0.781	0.609	0.163	0.258	0.200
7	train	1	0.657	0.432	0.823	0.678	0.183	0.261	0.216
8	test	1	0.822	0.676	0.765	0.585	0.163	0.250	0.197
8	train	1	0.669	0.447	0.809	0.654	0.180	0.256	0.211
9	test	1	0.858	0.736	0.759	0.576	0.020	0.143	0.035
9	train	1	0.728	0.529	0.795	0.632	0.037	0.143	0.059
10	test	1	0.822	0.675	0.766	0.586	0.163	0.238	0.193
10	train	1	0.667	0.445	0.810	0.656	0.180	0.248	0.209
15	test	1	0.819	0.670	0.769	0.592	0.163	0.251	0.198
15	train	1	0.663	0.440	0.813	0.660	0.180	0.258	0.212
20	test	1	0.868	0.753	0.803	0.645	0.029	0.143	0.049
20	train	1	0.766	0.587	0.835	0.698	0.054	0.143	0.078
30	test	1	0.868	0.753	0.760	0.578	0.020	0.143	0.035
30	train	1	0.766	0.587	0.789	0.623	0.035	0.143	0.057
60	test	1	0.852	0.726	0.865	0.748	0.100	0.105	0.103
60	train	1	0.750	0.562	0.879	0.773	0.121	0.114	0.118
100	test	1	0.868	0.753	0.752	0.566	0.019	0.143	0.033
100	train	1	0.766	0.587	0.781	0.610	0.033	0.143	0.054

Letter Recognition

Table 7.16: Performance for the letterrecognition dataset (stacking - metaclassifier-mlp100-0.02)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	1	0.446	0.199	0.977	0.954	0.843	0.460	0.595
1	train	1	0.424	0.180	0.978	0.956	0.859	0.475	0.611
2	test	1	0.437	0.191	0.977	0.954	0.778	0.455	0.574
2	train	1	0.413	0.170	0.978	0.956	0.788	0.468	0.587
3	test	1	0.448	0.200	0.976	0.953	0.736	0.437	0.549
3	train	1	0.430	0.185	0.977	0.955	0.708	0.449	0.549
4	test	1	0.445	0.198	0.976	0.953	0.811	0.423	0.556
4	train	1	0.430	0.185	0.977	0.954	0.817	0.432	0.566
5	test	1	0.428	0.183	0.977	0.955	0.845	0.511	0.637
5	train	1	0.396	0.157	0.979	0.958	0.856	0.531	0.655
6	test	1	0.453	0.205	0.977	0.954	0.713	0.459	0.559
6	train	1	0.429	0.184	0.977	0.955	0.755	0.470	0.579
7	test	1	0.435	0.189	0.978	0.956	0.834	0.477	0.607
7	train	1	0.409	0.167	0.978	0.957	0.837	0.494	0.621
8	test	1	0.452	0.204	0.977	0.954	0.801	0.453	0.578
8	train	1	0.433	0.187	0.978	0.956	0.804	0.466	0.590
9	test	1	0.459	0.210	0.976	0.953	0.810	0.470	0.595
9	train	1	0.433	0.187	0.977	0.954	0.850	0.485	0.618
10	test	1	0.436	0.191	0.977	0.955	0.903	0.479	0.626
10	train	1	0.412	0.169	0.978	0.957	0.918	0.493	0.642
15	test	1	0.448	0.201	0.975	0.951	0.700	0.475	0.566
15	train	1	0.427	0.183	0.976	0.953	0.714	0.493	0.583
20	test	1	0.410	0.168	0.976	0.953	0.807	0.519	0.632
20	train	1	0.383	0.147	0.978	0.956	0.829	0.541	0.655
30	test	1	0.449	0.202	0.972	0.945	0.593	0.500	0.543
30	train	1	0.430	0.185	0.973	0.947	0.600	0.511	0.552
60	test	1	0.525	0.275	0.970	0.940	0.615	0.428	0.505
60	train	1	0.513	0.263	0.970	0.941	0.616	0.432	0.508
100	test	1	0.658	0.434	0.967	0.934	0.505	0.306	0.381
100	train	1	0.641	0.411	0.967	0.935	0.523	0.319	0.396

Landsat

Table 7.17: Performance for the landsat dataset (stacking - metaclassifier-mlp30-0.07)

size	test/train	runs	misc	σ^2	ma acc	σ^2	ma prec	ma rec	ma F1
1	test	1	0.606	0.368	0.883	0.780	0.327	0.313	0.320
1	train	1	0.462	0.214	0.904	0.817	0.350	0.318	0.333
2	test	1	0.810	0.656	0.860	0.740	0.146	0.266	0.189
2	train	1	0.648	0.420	0.887	0.787	0.203	0.273	0.233
3	test	1	0.813	0.660	0.791	0.625	0.128	0.271	0.174
3	train	1	0.651	0.423	0.835	0.696	0.176	0.275	0.214
4	test	1	0.814	0.662	0.795	0.631	0.156	0.264	0.196
4	train	1	0.655	0.429	0.835	0.698	0.185	0.268	0.219
5	test	1	0.814	0.662	0.769	0.591	0.151	0.264	0.192
5	train	1	0.656	0.431	0.813	0.662	0.177	0.269	0.214
6	test	1	0.819	0.670	0.772	0.595	0.164	0.243	0.196
6	train	1	0.664	0.441	0.814	0.663	0.182	0.250	0.210
7	test	1	0.816	0.666	0.781	0.609	0.163	0.258	0.200
7	train	1	0.657	0.432	0.823	0.678	0.183	0.261	0.216
8	test	1	0.822	0.676	0.765	0.585	0.163	0.250	0.197
8	train	1	0.669	0.447	0.809	0.654	0.180	0.256	0.211
9	test	1	0.858	0.736	0.759	0.576	0.020	0.143	0.035
9	train	1	0.728	0.529	0.795	0.632	0.037	0.143	0.059
10	test	1	0.822	0.675	0.766	0.586	0.163	0.238	0.193
10	train	1	0.667	0.445	0.810	0.656	0.180	0.248	0.209
15	test	1	0.819	0.670	0.769	0.592	0.163	0.251	0.198
15	train	1	0.663	0.440	0.813	0.660	0.180	0.258	0.212
20	test	1	0.868	0.753	0.803	0.645	0.029	0.143	0.049
20	train	1	0.766	0.587	0.835	0.698	0.054	0.143	0.078
30	test	1	0.868	0.753	0.760	0.578	0.020	0.143	0.035
30	train	1	0.766	0.587	0.789	0.623	0.035	0.143	0.057
60	test	1	0.852	0.726	0.865	0.748	0.100	0.105	0.103
60	train	1	0.750	0.562	0.879	0.773	0.121	0.114	0.118
100	test	1	0.868	0.753	0.752	0.566	0.019	0.143	0.033
100	train	1	0.766	0.587	0.781	0.610	0.033	0.143	0.054

Magic

Chapter 8

Discussion, Conclusions and Further Work

This empirical sampling of the performance of different ensemble techniques leads to some interesting, and perhaps not very surprising, observations. In general these are mostly confirmations of concepts that are already known about ensemble learning.

8.1 Validity of results

One thing that needs to be considered is that the limited amount of datasets used is very small and certainly not sufficient to consider any results as proof, but they are certainly an indicator of the performance improvements that could be achieved with more sofisticated machine learning techniques.

8.2 Efficiency

The efficiency of training ensemble solutions, especially when the technique involves sequential training of each member, is something that needs to be considered when picking ensemble techniques. The time required for training tends to grow proportionally to the size of the ensemble and the complexity of the architecture. This is especially true when considering online training of ensemble techniques.

8.3 Noise

Another benefit that can be observed is the reduction of noise in the classification decisions over repeated tests. Intuitively we can imagine that when there are many components that need to reach an agreement it is less likely that there will be many outliers.

8.4 Different datasets have different results

It is evident from observation of the results that different datasets will produce varied results, this is mainly due to the fact that the multi layer perceptrons used in the empirical tests might not be a good method for those specific datasets. Different datasets also have different applicability to an ensemble learning technique, and generally we notice that datasets that require a "deeper" learning seem to produce results that are not as good (but still better than the single classifier).

8.5 Further work

Although this was a very long project, the entirety of the work done in this period is only scratching the surface of what else could be done.

8.5.1 Re-running benchmarks with different parameters

Despite the amount of care that has gone into the selection of parameters, it has been found in a few cases that there was either underfitting or overfitting. It would be ideal, given the time, to repeat these tests with new parameters, and using alternative neural network architectures and ml techniques.

8.5.2 Further expansions to Encog

Regression

All these techniques have been implemented and studied with a regard towards classification problems. It should be sufficiently easy to adapt the source code to support regression as well as classification. In some places I have already inserted placeholders or comments for what could be done to achieve this, so that before contributing my work back I can add this functionality.

Other techniques

The techniques implemented in this paper are not the only ensemble techniques. There are several others, all of which could be implemented taking advantage of the basic framework that I have put together. This is left as a further development by the community.

8.5.3 Contribution back to the community

Due to the time constraints, it was not possible to prepare the source code in the form of a patch that could be contributed back to the Encog community. This is a process that I intend to complete within a short time after the submission of this document and all that is required is to produce a patch set that does not contain any of the training experiments, as well as writing the relevant javadoc entries and generic documentation.

8.5.4 Real-world use of Encog Ensembles

This study lacks coverage of real world applications on datasets that have not been already extensively used. Amongst future plans, I intend to validate the use of ensembles in Encog by applying some techniques to problems and competitions openly available. This should provide more tangible proof of the usefulness of these methods.

Bibliography

[1] Vijaya Kumar B Al-Ghoneim K. Learning ranks with neural networks. *Application and Science of Artificial Neural Networks: proceedinges of the SPIE*, 2492:446–464, 1995.

[2] M Anthony and P Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 1999.

[3] R Beale and T Jackson. *Neural Computing - an introduction*. Institute of Physics Publishing, 1990.

[4] C Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[5] C Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

[6] R K Bock, A Chilingarian, A Gaug, M Hakl, F Hengstebeck, T Jirina, M Klaschka, J Kotrc, P Savicky, S Towers, A Vaicilius, and W Wittek. Methods for multidimensional event classification: a case study using images from a cherenkov gamma-ray telescope. *Nucl. Instr. Meth. A*, 516:511–528, 2004.

[7] D Brain and G I Webb. On the effect of data set size on bias and variance in classification learning. *Proceedings of the Fourth Australian Knowledge Acquisition Workshop*, pages 117–128, 1999.

[8] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[9] Dasarathy B.V. and Sheela B.V. A composite classifier system design: Concepts and methodology. *Proceedings of the IEEE*, 67(5):708 – 713, may 1979.

[10] S. Cho and J. Kim. Multiple network fusion using fuzzy logic. *IEEE Transactions on Neural Networks*, 6(2):497 – 501, March 1995.

[11] Nguyen D and Widrow B. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. *IJCNN International Joint Conference on Neural Networks*, pages 21–26, 1990.

[12] Opitz D and Shavlik J. Generating accurate and diverse members of a neural-network ensemble. *NIPS*, 8:535–541, 1996.

[13] Wolpert D. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[14] J Dvorak and P Savicky. Softening splits in decision trees using simulated annealing. *Proceedings of ICANNGA 2007, Warsaw*, pages 721–729.

[15] P W Frey and D J Slate. Letter recognition using holland-style adaptive classifiers. *Machine Learning*, 6(2), March 1991.

[16] G. Fumera and F. Roli. Performance analysis and comparison of linear combiners for classifier fusion. 2002.

[17] Giacinto G, Perdisci R, Del Rio M, and Roli F. Intrusion detection in computer networks by a modular ensemble of one class classifiers. *Information Fusion*, 2006.

[18] E Gamma, R Helm, R Johnson, and J Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] S Geman, E Bienenstock, and R Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–48, 1992.

[20] X Giannakopoulos, J Karhunen, and E Oja. An experimental comparison of neural algorithms for independent component analysis and blind separation. *Int. J. Neural Syst*, 9, 1999.

[21] C Goutte and E Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. 2005.

[22] S J Haberman. Generalized residuals for log-linear models. *Proceedings of the 9th Internation Biometrics Conference, Boston*, pages 104–122, 1976.

[23] Srihari S Ho T, Hull J. Decision combination in multiple classifier systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:66–76, 1994.

[24] Chuanyi Ji and Sheng Ma. Combinations of weak classifiers. *IEEE Transactions on Neural Networks*, 8(1):32–42, January 1997.

[25] J Kittler, M Hatef, R Duin, and J Matas. On combining classifiers. *IEEE Transactions on pattern analisys and machine intelligence*, 20(3):226 – 239, March 1998.

[26] L. Kuncheva. A theoretical study on six classifier fusion strategies. *IEEE Transactions on pattern analisys and machine intelligence*, 24(2):281 – 286, February 2002.

[27] J M Landwehr, D Pregibon, and A C Shoemaker. Graphical models for assessing logistic regression models (with discussion). *Journal of the American Statistical Association*, 79:61–83.

[28] S Lawrence, I Burns, A Back, A Chung Tsoi, and C L Giles. *Neural Network Classification and Prior Class Probabilities*, volume 1524. 1998.

[29] G Lera. Neighborhood based levenberg-marquardt algorithm for neural network training. 2002.

[30] W D Lo. *Logistic Regression Trees*. PhD thesis, University of Wisconsin, 1993.

[31] CD Manning, P Raghavan, and H Schutze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[32] M Moller. A scaledconjugategradient algorithm for fast supervised learning. 1993.

[33] Melville P and Mooney R. Creating diversity in ensembles using artificial data. *Information Fusion*, 6:99–111, 2005.

[34] Quinlan R. Bagging, boosting, and c4.5. 2006.

[35] Yuval Raviv and Nathan Intrator. Bootstrapping with noise: An effective regularization technique. *Connection Science*, 8:355–372, 1996.

[36] M Riedmiller. Rprop - description and implementation details. 1994.

[37] R Rojas and J Feldman. *Neural Networks: A Systematic Introduction*. Prentice Hall, 1996.

[38] S Russell and P Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1998.

[39] P Savicky and E Kotrc. Experimental study of leaf confidences for random forest. *Proceedings of COMPSTAT 2004, In: Computational Statistics*, pages 1767–1774, 2004.

[40] R. E. Schapire. The strength of weak learnability. *Machine Learning*, 5:197–227, 1990.

[41] R. E. Schapire and Y Freund. Experiments with a new boosting algorithm. *Machine Learning: proceedings of the Thirteenth International Conference*, pages 148–156, 1996.

[42] R. E. Schapire and Y Freund. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[43] R E Schapire, Y Freund, P Bartlett, and W Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. 1998.

[44] R. E. Schapire, Y Freund, and R Iyer. An efficient boosting algorithm for combining preferences. *Machine Learning: proceedings of the Fifteenth International Conference*, 1998.

[45] K Tumer and J Ghosh. Robust combining of disparate classifiers through order statistics. Technical report, AT&T Labs, 1999.

[46] K Tumer and N C Oza. Decimated input ensembles for improved generalization. Technical report, NASA Ames Research Center, 1999.

[47] Wikipedia. Precision and recall. `http://en.wikipedia.org/wiki/Precision_and_recall`, 2012. [Online; Accessed Aug 31st, 2012].

[48] Shuang Yang and Antony Browne. Neural network ensembles: combining multiple models for enhanced performance using a multistage approach. *Expert Systems*, 21(5):279–288, November 2004.