AN ABSTRACT OF THE THESIS OF

Saket Subhash Joshi for the degree of Master of Science in Computer Science presented on October 3, 2003.

Title:

Calibrating Recurrent Sliding Window Classifiers for Sequential Supervised Learning

Abstract approved: [redacted]

Thomas Dietterich

Sequential supervised learning problems involve assigning a class label to each item in a sequence. Examples include part-of-speech tagging and text-to-speech mapping. A very general-purpose strategy for solving such problems is to construct a recurrent sliding window (RSW) classifier, which maps some window of the input sequence plus some number of previously-predicted items into a prediction for the next item in the sequence. This paper describes a general-purpose implementation of RSW classifiers and discusses the highly practical issue of how to choose the size of the input window and the number of previous predictions to incorporate. Experiments on two real-world domains show that the optimal choices vary from one learning algorithm to another. They also depend on the evaluation criterion (number of correctly-predicted items versus number of correctly-predicted whole sequences). We conclude that window sizes must be chosen by cross-validation. The results have implications for the choice of window sizes for other models including hidden Markov models and conditional random fields.

Calibrating Recurrent Sliding Window Classifiers for Sequential Supervised Learning

by

Saket Subhash Joshi

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented October 3, 2003
Commencement June 2004

Master of Science thesis of Saket Subhash Joshi presented on October 3, 2003

APPROVED:

[redacted]

Major Professor, representing Computer Science

[redacted]

Director of the School of Electrical Engineering and Computer Science

[redacted]

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

[redacted]

Saket Subhash Joshi, Author

ACKNOWLEDGMENTS

Dedicated to my Father.

I would like to take this opportunity to thank my advisor, Thomas Dietterich, for his invaluable ideas and constructive criticism. I would also gratefully acknowledge the support of the National Science Foundation under grants IIS-0083292 and ITR-0001197.

TABLE OF CONTENTS

Page

Chapter 1: Introduction 1

1.1 Sequential Supervised learning 1

1.2 The 8-Case Analysis for sequential supervised learning problems 4

1.3 Popular sequential supervised learning algorithms 5

1.3.1 Hidden Markov Models 6
1.3.2 Maximum Entropy Markov Models 6
1.3.3 Conditional Random Fields 7
1.3.4 Sliding Windows and Recurrent Sliding Windows 8

1.4 Our Approach 9

Chapter 2: Previous Work 13

2.1 NETtalk (text-to-speech mapping) 13

2.2 Prediction of protein secondary structure 14

2.3 Part-of-speech tagging 15

2.4 Information Extraction 16

2.5 Summary 18

Chapter 3: Experiments and Results 19

3.1 Experimental Methods: 19

3.2 Results 21

Chapter 4: Discussion 29

TABLE OF CONTENTS (Continued)

		Page
Chapter 5:	Conclusions and Future work	38
Bibliography		40
Appendix		42

LIST OF FIGURES

Figure | Page

1.1 Pictorial view of the 8-Case Analysis 4

1.2 Hidden Markov Model 5

1.3 Max Entropy Markov Model 5

1.4 Conditional Random Fields 5

3.1 NETTalk: % of individual letters correct 21

3.2 Protein Secondary Structure: % of individual residues (amino acids) correct 22

3.3 NETTalk: % of whole words correct 23

3.4 Contour plot showing Bagging performance as a function of input and output context. 24

3.5 Contour plot showing boosting performance as a function of input and output context. 24

3.6 NETTalk: % of individual letters correct as a function of output context 25

3.7 Protein Secondary Structure: % of individual residues (amino acids) correct as a function of output context 26

3.8 NETTalk: % of whole words correct as a function of output context 27

4.1 A Case Analysis of Error for 2 classes 31

4.2 A Case Analysis of Error for multiple classes 32

4.3 Bias Variance Analysis of J48 against output window size with IC=0 for NETtalk data 33

4.4 Bias Variance Analysis of Bagged J48 against output window size with IC=0 for NETtalk data 34

4.5 Bias Variance Analysis of J48 and Bagging as a function of input window size with OC=0 for NETtalk data 35

LIST OF TABLES

Table — Page

1.1 The 8-Case Analysis 3

1.2 Simple sliding windows 9

1.3 Recurrent sliding windows 9

LIST OF APPENDIX FIGURES

Figure | Page

A.1 The filters sub-package 45

A.2 The classifiers sub-package 48

CALIBRATING RECURRENT SLIDING WINDOW CLASSIFIERS FOR SEQUENTIAL SUPERVISED LEARNING

CHAPTER 1

INTRODUCTION

1.1 Sequential Supervised learning

The standard supervised learning problem is to learn to map from an input feature vector x to an output class variable y given N training examples of the form $(x_i, y_i)_{i=1}^{N}$. Problems where the variable y can take up a finite number of discrete values are called classification problems in the literature. Standard supervised learning algorithms like decision tree algorithms, the naive Bayes algorithm, the K-nearest neighbor algorithm etc., learn this map and come up with a classifier which given a new example feature vector x_{new} as input, produces the corresponding class value y as output. The accuracy of a classifier is measured as the percentage of such new data points (data points not included in the set of training examples) it classifies correctly.

Many problems, however, do not succumb to this simple approach. Consider the problem of part-of-speech tagging. The problem can be defined as follows. Given a set of sentences in the English language, label each word according to the part of speech it represents in the sentence. Trying the classical supervised learning approach on this problem would lead to construction of a dataset where each example would be a word broken down into a number of features and the class variable specifying the corresponding part of speech (e.g., noun, verb, adjective, etc.). We could build a classifier by feeding some training data (data where we already know the correct part of speech for each word) to a standard learning algorithm and then classify new words. This method would not work well however, as the problem arises with words like "lead". There is no way to tell if the word "lead" viewed in isolation is an adjective as in "a lead pencil", or a verb as in the sentence "please lead the way". The classification here depends on the relationship of this word to the other words in the sentence. The classification is bound to the "sequence" in which this word appears. Therefore the part-of-speech can be seen as a problem of mapping input sequences to output sequences. No learning algorithm can be accurate in this setting unless it has a way to analyze the context around individual words. A standard Machine Learning approach would try to process each word separately, but this example shows that would not work because at a minimum each word needs to be handled in context. We need algorithms to learn a map from input sequences to output sequences.

Many recent learning problems can be viewed as extensions of standard supervised learning to the setting where each input object X is a *sequence* of feature vectors, $X = \langle x_1, x_2, \ldots, x_T \rangle$, and the corresponding output object Y is a sequence of class labels $Y = \langle y_1, y_2, \ldots, y_T \rangle$. The *sequential supervised learning* (SSL) problem is to learn to map from X to Y given a set of N training examples, $(X_i, Y_i)_{i=1}^{N}$.

TABLE 1.1: The 8-Case Analysis

000	(a), (b), and (c) are all random. There is no point in learning.
001	(a) and (b) are random but there is a pattern in (c). Just learn the $x \rightarrow y$ mapping.
010	(a) and (c) are random. Just model y, e.g., as a Markov chain or time series.
011	(a) is random. This is the HMM case (HMMs are discussed in section 1.3.1). Model y as a Markov chain, model $y \rightarrow x$ and then invert using Bayes' theorem. Alternatively, we can try to use a recurrent sliding window (discussed in section 1.3.4) that has an input context of 0 but a large recurrent context. Ideally we would use information from the entire sequence Y (as in CRFs and HMMs). However this is expensive, so we will use "half" of the information (just one-sided output context)
100	(b) and (c) are random. Just model x as a Markov chain or time series.
101	(b) is random. Model the x's as a Markov chain and learn $x \rightarrow y$ mapping. The simple sliding window method (discussed in section 1.3.4) with a big input context and no output context is appropriate here.
110	(c) is random. Model x's as a Markov chain and the y's as an independent Markov chain. Again, there is no point in learning here.
111	All three relationships are important. In this case we need to capture the sequential relationships in both the x's and the y's as well as the relationships between the x's and the y's. CRFs are one approach; recurrent sliding windows with big input context and big output context are another approach.

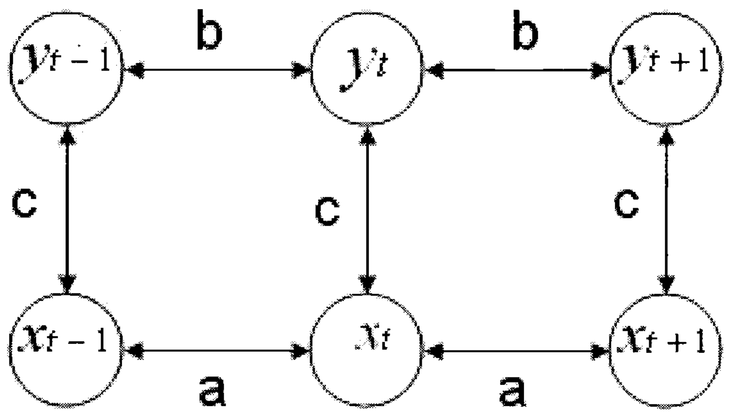

FIGURE 1.1: Pictorial view of the 8-Case Analysis

1.2 The 8-Case Analysis for sequential supervised learning problems

It is hard to predict an output sequence given an input sequence directly in a single step. The number of possible output sequences is exponential in the length of the sequences, and classification methods do not work well with large numbers of classes. Therefore we seek a divide-and-conquer approach. There are three relationships that are important in a sequential supervised learning problem:

(a) The sequential relationships among the x's

(b) The sequential relationships among the y's

(c) The mapping from x's to y's

We can imagine 8 cases as shown in table 1.1. Because we are interested in cases where X provides some information about Y, we can ignore cases 000, 010, 100, and 110. Case 001 can be handled by standard supervised learning. Cases 011, 101 and 111 can be handled by sliding windows, recurrent sliding windows, or some other popular sequential supervised learning algorithms. In the next section we will see some such popular sequential supervised learning algorithms that exploit the relationships (a), (b), and (c) in data.

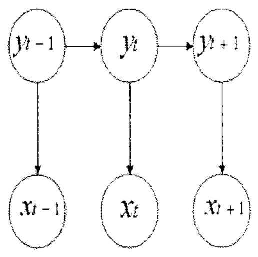

FIGURE 1.2: Hidden Markov Model

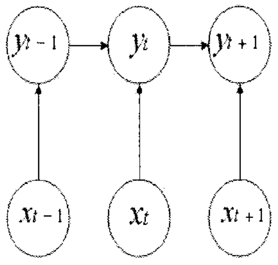

FIGURE 1.3: Max Entropy Markov Model

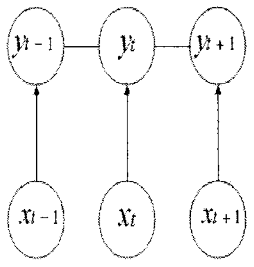

FIGURE 1.4: Conditional Random Fields

1.3 Popular sequential supervised learning algorithms

In the literature, two general strategies for solving SSL problems have been studied. One strategy, which we might call the direct approach, is to develop probabilistic models of sequential data. The advantage of these probabilistic models is that they seek to capture the true sequential relationships that generate the data. Examples of such strategies are hidden Markov models, maximum entropy Markov models, and conditional random fields. The other general strategy that has been explored might be called the Indirect approach (i.e., a "hack"). In this strategy, the sequential supervised learning problem is solved indirectly by first converting it into a standard supervised learning problem, solving that problem, and then converting the results into a solution to the SSL problem. Examples of indirect approaches include sliding windows and recurrent sliding windows. Let us look at each of them with more emphasis on sliding windows and recurrent sliding windows because they are the basis for this thesis.

1.3.1 Hidden Markov Models

The hidden Markov model (Rabiner, 1989) is a generative model of the joint distribution $P(X, Y)$ of the object sequence X and the label sequence Y. Dynamic Bayesian networks can use information from previous time steps (or neighboring examples in the sequence) for current evaluation. Dynamic Bayesian networks factor the state space into subspaces. This is especially useful when the interactions between the subspaces is sparse. If the subspaces are highly correlated, it is more useful to combine them and treat them as a single space. A hidden Markov model is a special case of dynamic Bayesian networks that does not factor the state space. Therefore it is used in problems where such factoring is superfluous. Hidden Markov models are ideally suited to solve problems that can be classified as case 011. Figure 1.2 shows an example of a HMM. The popular forward-backward prediction algorithm provides efficient inference for HMMs. We will see examples of real world problems currently solved using HMMs in the next chapter.

1.3.2 Maximum Entropy Markov Models

The HMM is a generative model, which means that it models the relationship (c) as a map from y to x. During classification, Bayes' rule is employed to infer the

y values given the x values. There are two potential problems with this method. Firstly a joint distribution of the object sequence and the label sequence has to be maintained because of which, the required resources grow exponentially with the number of input features. Secondly a generative models may not exactly model the causal relationships between the x's and the y's correctly. In some problems the causal relationship is from the x's to the y's. Maximum entropy Markov models (McCallum, Freitag & Pereira, 2000), which are another sequential supervised learning device, model exactly that relationship. MEMMs learn conditional probability tables rather than a joint distribution. This avoids problems caused by a large input feature space. MEMMs require only simple extensions to the forward-backward algorithm. Figure 1.3 shows an example of a MEMM.

1.3.3 Conditional Random Fields

More recently, the conditional random field (Lafferty, McCallum, & Pereira, 2001) has been proposed as a model of the conditional distribution $P(Y|X)$. It is another sequential supervised learning approach that goes beyond MEMMs in that it works with a conditional model and additionally solves the label bias problem that is caused by the use of the exponential transition model of MEMMs. In MEMMs, the probabilities of state transitions are normalized locally for each time step rather than normalizing over the entire sequence of y's. As a result, they do not compute the globally most likely sequence. This means that the probabilities of transitions from state y_t to state y_{t+1} are normalized over all possible states y_{t+1} that can be reached from y_t. As a result, probabilities of transitions to states with fewer outgoing states increase. This gives rise to a bias towards states with fewer number of possible next states, and this is termed as the label bias problem. CRFs solve this problem by employing global normalization. CRFs can be pictured as Markov random fields of the y's conditioned on the x's. An example of CRFs is shown in Figure 1.4. Conditional random fields constitute a very promising approach, and they are attracting further applications and research.

1.3.4 Sliding Windows and Recurrent Sliding Windows

The indirect approach employed in sliding windows and recurrent sliding windows is to convert the input and output sequences into a set of windows as shown in Table 1.2. Each window consists of central element x_i and LIC letters of left input context and RIC letters of right input context (in the figure $LIC = RIC = 3$). Contextual positions before the start of the sequence or after the end of the sequence are filled by a designated null value (in this case _).

In most SSL problems, there are regularities in the sequence of y values. In part-of-speech tagging, the grammar of natural language constrains the possible sequences of parts of speech. In text-to-speech mapping, there are patterns in the phoneme sequence. The simple sliding window method cannot capture these patterns unless they are completely manifested in the X values as well, which is rarely the case. One way to partially learn these patterns is to employ *recurrent* sliding windows, in which previous predictions (e.g., for y_{t-1}, y_{t-2}, etc.) are fed back as input features to help predict y_t. During learning, the observed labels in the training set can be used in place of these fed back values. During classification, the sequence is processed from left-to-right, and the

TABLE 1.2: Simple sliding windows

(X,Y)	enough	In^-f-
w_1	___enou	I
w_2	__enoug	n
w_3	_enough	^
w_4	enough_	-
w_5	nough__	f
w_6	ough___	-

TABLE 1.3: Recurrent sliding windows

(X,Y)	enough	In^-f-
w_1	___enou__	I
w_2	__enoug_I	n
w_3	_enoughIn	^
w_4	enough_n^	-
w_5	nough__^-	f
w_6	ough___-f	-

predicted outputs $\hat{y}_{t-2}, \hat{y}_{t-1}$ are fed as input features to predict y_t. Note that the sequence could also be processed from right-to-left, but not simultaneously left-to-right and right-to-left. We denote the number of fed back y values as the left output context LOC or right output context ROC, depending on the direction of processing. Table 1.3 shows the training windows with $LOC = 2$ and $ROC = 0$. The advantage of the recurrent sliding window approach is that it can be combined with *any* standard supervised learning algorithm to solve SSL problems.

1.4 Our Approach

Two practical issues arise in applying either the direct or indirect methods. The first issue is the size of the context. How much of the input sequence and output sequence should be considered at each point along the sequence? For the recurrent sliding window methods, this can be stated concretely as follows: how large should the left input context, right input context, and left output context be? Presumably larger contexts will lead to high variance and overfitting by the learning algorithm, whereas smaller contexts will lead to lower variance but higher bias and underfitting by the learning algorithm.

Based on the 8-case analysis, we can say that only in case 111 do we expect there to be a trade-off between input and output contexts. In the 101 case, output context should always be 0. In the 011 case input context should always be 0. So in these two cases there is no trade-off: one form of context is simply useless. This may explain some of our results (or at least it may suggest a way of interpreting our results).

The second issue is the criteria for measuring performance. A straightforward generalization of the 0/1 loss for standard supervised learning is to measure the number of whole sequences that are correctly predicted. However, another measure is the number of individual x_t items correctly classified (e.g. individual words in part-of-speech tagging). An interesting question is whether the optimal choice of input and output context depends on the performance criterion. Let us analyze what the 8-case analysis says about this.

001: The two criteria should be the same.

011: If we use a very large output context, this will make the whole word predictions more accurate, but it may also increase variance. So we might predict that a smaller output context would be more appropriate for individual letter predictions, since whole word accuracy is less important and this pushes us to reduce variance by using a smaller output context.

101: Again we predict that for whole word accuracy we want larger intput context. So we predict that a smaller context would be appropriate for single letter predictions.

111: For the whole word criterion, this would suggest larger output context (which, because of the trade-off means smaller input context). For the single letter criterion it depends on which influence is stronger. If the $y \rightarrow y$ correlations are weaker than the $x \rightarrow x$ correlations, then we use a larger input context, otherwise a larger output context.

In order to test our predictions we conducted an experimental study of these two issues. In this thesis, we present the experimental study employing a general-purpose recurrent sliding window system, RSW, that we have constructed as part of the WEKA, the widely used Java based Machine Learning library developed at the University of Waikato, NZ. RSW can work with any of the WEKA classifiers that return class probability distributions[1]. Our study compares naive Bayes, decision trees, bagged trees, and adaboosted trees on two

[1] Interested readers can find details about the implementation and documentation of RSW in the appendix

SSL problems: NETtalk and protein secondary structure prediction. We find that our intuitions are somewhat justified. There is some trade-off between the amount of input context and the amount of output context for the NETtalk data (which clearly belongs to case 111), but it is not crystal clear. Furthermore, we show that better learning algorithms (e.g., adaboosted trees) are able to handle larger amounts of both input and output context. Finally, we show that the appropriate choice of input and output context depends on the learning algorithm and on the evaluation criterion. This suggests that there are no general-purpose methods for choosing the amount of input and output context in an algorithm-independent way.

CHAPTER 2

PREVIOUS WORK

At this point we will present an introduction of some previously studied sequential supervised learning problems and approaches that have been used to solve them.

2.1 NETtalk (text-to-speech mapping)

The NETtalk problem is defined as follows. Given a list of words and their corresponding phoneme-stress sequences, learn a map that can be applied to predict the correct phoneme-stress sequence of a new word. A phoneme is the most basic unit of speech and a stress is a measure of emphasis given to that phoneme during pronunciation. The phoneme and stress pair together form a label which can be fed to a speech synthesis system to generate spoken words. Since the words themselves can be viewed as a sequence of letters, NETtalk can be viewed as a sequential supervised learning problem in which the x's are the individual letters and the y's are the corresponding phoneme-stress pairs (class labels). Some of the early work with sequential data using sliding windows was done for the NETtalk system (Sejnowski & Rosenberg, 1987). The NETtalk system learnt to pronounce English text by learning a map from individual letters in a word to their respective phonemic sound symbols coupled with the amount of stress associated with that phoneme. An example is given in Tables

1.2 and 1.3 in the previous chapter. Sejnowski & Rosenberg (1987) used a neural network with 203 input units, 80 hidden units, and 26 output units. The input units were divided into sets of 29 units each (26 letters and 3 extra features). Each feature was a single bit that was 1 if a particular letter appeared at a particular position in the input. Each of these 7 sets represented a letter and altogether a 7 letter window. The NETtalk system achieved a performance of around 93% on the training data for predicting individual letters correctly and around 65% for predicting whole words correctly. However this system was specifically built and tuned to solve the text-to-speech problem alone. An interesting inference with the NETtalk data that was indicated by Sejnowski & Rosenberg, and later clearly presented by Bakiri & Dietterich (1991), was that the pronunciation (phoneme and stress) of a letter depends more on the letters that come after it than on those that come before it. As a consequence, a window sliding over the word from right to left gives better performance than a window sliding in the opposite direction. Later Adamson and Damper (1996), employed a similar approach but with a recurrent network on this problem of learning to pronounce English text. We have used the NETtalk data for our experiments.

2.2 Prediction of protein secondary structure

An important problem in molecular biology is that of predicting "protein secondary structure". Briefly, the task is as follows. For each element in a protein sequence (drawn from a 20-letter alphabet), assign it to one of three classes (alpha helix, beta sheet, or coil). If we view the class labels or residues (alpha helix, beta sheet, or coil) as a sequence corresponding to the protein sequence, then this task can be formulated as a sequential supervised learning problem where the x's represent the proteins and the y's represent the individual class label (alpha helix, beta sheet, or coil). As a result of the sequential nature of protein and DNA data, applications of sequential data classifiers in Computational Biology are immense. Qian & Sejnowski (1988) employed a sliding window of width 15 with neural networks in their work on prediction of secondary structure of globular proteins. Later Krogh et al. (1993) used an HMM to model protein families. Again the attempts here have been to come up with a model specifically tuned to solve this problem. We have used the Qian & Sejnowski data sets for training and testing of prediction of secondary structure of proteins as well, in our experiments.

2.3 Part-of-speech tagging

Part-of-speech tagging has been defined in the previous chapter. For more than thirty years POS tagging has been a standard for the sequential data learning task. Yet solutions to this problem have been ad-hoc. Words that can potentially belong to more than one part of speech category are called ambiguous words in the literature. Researchers have figured out to a large extent what the unambiguous words are and have built dictionaries for them. The majority of the research in POS tagging is conducted for disambiguation given the context in which the word appears. The context is generally selected as a few words appearing before and after the current word in the sentence. Greene & Rubin (1971) used a rule-based approach to build a text tagger and achieved up to 77% accuracy on test data. Later improvements were made, and statistical methods started finding their way into this area quite early. Jelinek (1985) used a HMM for this purpose. Almost all work on POS tagging after that has been with improving HMM performance for this problem. Cutting, Kupiec et al. (1992) also used a HMM for their POS tagging system and employed ambiguity classes (classes of ambiguous words) and a phrase recognition system based on word sequences to achieve an accuracy of 96% on unseen data. Brill (1992) deviated from the main stream and introduced a trainable rule-based tagger. However his system too used a lot of sequential information and did equally well (96.5% accuracy). Ratnaparkhi (1996) inserted a bunch of extra features for rarely occuring and ambiguous words. To avoid feature space explosion, he used a maximum entropy Markov model. This was an ad-hoc strategy meant specifically to improve performance on the POS tagging problem. Recently Marquez & Padro (2000) presented a sliding window approach with an output context of 5. They divided the POS tagging problem into many classification problems, one for each ambiguity class and induced statistical decision trees to model grammar constraints (accuracy 99%). Lafferty, McCallum & Pereira (2001) employed the POS tagging domain to present Conditional Random Fields. POS tagging has application in language understanding and information extraction (a problem in itself for which state-of-the-art systems learn sequential patterns of text).

2.4 Information Extraction

The field of information extraction (IE) deals with segmenting and extracting information from documents. For example, we might wish to extract information from web pages such as names, address, and title of the CEO of a company (extracted from a corporate webpage). Another example would be to extract the authors, title, journal, year, and page numbers of a published article by analyzing the citations in a paper. Since it is more likely that the name of the author would follow a word like "Author" or a phrase like "Written by" in the document, there is a possibility of extracting information from the sequence in which words appear. IE can therefore be modeled as a sequential supervised learning problem where the x's represent the words in the document and the y's represent a binary class (e.g., "author" and "not author"). Classical IE systems applied NLP as their core. Statistical methods were employed in IE systems like FASTUS 1993, and CIRCUS 1993. Cohen (1995) presented text classification using relational learning with ILP methods and Freitag (1996) proposed the use of Machine Learning methods with IE where text formats are fairly informal along with the use of sliding windows and HMMs for learning sequential text patters. Later HMMs started coming into use more frequently in IE (Leek 1997) and Craven et.al (1998) suggested the use of relational learning methods that learn to classify Web pages on the basis of the sub-graph of pages surrounding a given page. This was an application of IE where they identified useful fields in existing text (in their case, web pages). After that Freitag & McCallum (1999) implemented the use of HMMs to model the same for IE with a statistical technique called shrinkage which, they explain, brought in the best of both worlds in terms of the bias variance tradeoffs. More recently McCallum, Freitag & Pereira (2000) exhibited the use of maximum entropy Markov models for IE. This technique not only modeled the causal relationships in the problem better but also reduced the space of possible feature value combinations required to be enlisted. Sequential supervised learning techniques are commonplace and taken for granted in IE applications today.

2.5 Summary

Among other applications of such sequential data classifiers are handwriting recognition systems (Bengio et al. 1995, Hu & Brown 1996), Speech Recognition (Rabiner 1989), intrusion detection systems (Lee, Stolfo & Mok 1998), fraud detection systems (Fawcett & Provost 1997) and many more. Solutions to all these problems have been either ad-hoc or extremely complex. Since we can classify all the above problems as sequential supervised learning problems, we can imagine a generalized approach to solving them without using ad-hoc strategies. A general purpose tool is needed for such problems. Such a robust and efficient off-the-shelf implementation will not only aid in the development of new applications in these and other areas but also allow researchers to share a common platform for comparative study.

RSW is the first general purpose off-the-shelf classifier for sequential data using recurrent sliding windows. However, since a general purpose tool is, by definition, not tuned to solve any specific problem, the two issues discussed in section 1.4 arise. Both these issues can be summed up into one question: given a problem domain, how can we calibrate the window size for a recurrent sliding window classifier so as to achieve optimal performance? In this thesis along with RSW we also present the first systematic study of calibration of window size for RSW through our experimental study.

CHAPTER 3

EXPERIMENTS AND RESULTS

3.1 Experimental Methods:

For our experiments, we chose two large datasets. The first is the text-to-speech dataset from the NETtalk system (Sejnowski & Rosenberg 1987). This dataset consists of 20,003 words, expressed as sequences of letters and their corresponding pronunciations in terms of phonemic sound representations. Of these, 2000 sequences (words) were picked at random without replacement. This dataset of 2000 sequences was then randomly split into a 1000-sequence training dataset and a 1000-sequence test dataset. Each class label consists of a phoneme-stress pair. For example, in words like "there", the "t" is pronounced as `"T>"` wherein the "T" is the phoneme label and the ">" is the stress label. There are 56 phonemic sounds produced in English speech and 6 different levels of stress. Of all the 336 possible combinations of phonemes and stresses only 140 are observed in the training and test sets. The average length of English words in the data is 7 (minimum 2, maximum 17).

The second problem we studied was the Protein Secondary Structure Prediction dataset employed by Qian & Sejnowski (Qian & Sejnowski 1988). Their task was to assign each element in a protein sequence (drawn from a 20-letter alphabet) to one of three classes (alpha helix, beta sheet, or coil). Although there are only three classes, the average length of the protein sequences is 169 (minimum 11, maximum 498).

We computed two measures of performance:

(a) The percentage of individual elements in the test set predicted correctly and

(b) the percentage of whole sequences in the test set predicted correctly. For the latter, all elements in the sequence must be correctly predicted in order for the sequence to be declared correctly predicted.

Because the sequences are so long, we never observed a case where an entire protein sequence was correctly predicted.

For each dataset and each performance measure, we applied RSW with $LIC = RIC = 0 \ldots 7$. This resulted in an input window size ranging from 1 to 15 by steps of 2. For each input window size, experiments were performed with the right output context (ROC) varying from 0 to 7 and then with the left output context (LOC) varying from 0 to 7. This gives a total of 120 experiments with different input and output context combinations.

Each of the 120 experiments was performed with each of four learning algorithms: naive Bayes, J48[1] decision trees, bagged J48 trees, adaboosted J48 trees (boosting by weighting). For decision trees, we employed pessimistic pruning with five confidence factors (0.1, 0.25, 0.5, 0.75, 1). With bagged and boosted trees, we experimented with 10 to 50 unpruned trees.

[1] J48 is the WEKA implementation of C4.5 (Quinlan 1993)

3.2 Results

Figures 3.1, 3.2, and 3.3 compile the results of the above experiments. Each graph displays a comparison of the performance of various learning algorithms across various input contexts for the best configuration of the remaining parameters. A label next to each performance curve displays the name of the learning algorithm and the output context. The output context chosen for each curve is the one where the performance of the algorithm achieves its maximum. Where there is a difference in performance between the use of left and right output context, it is specified appropriately by an *L* or an *R*.

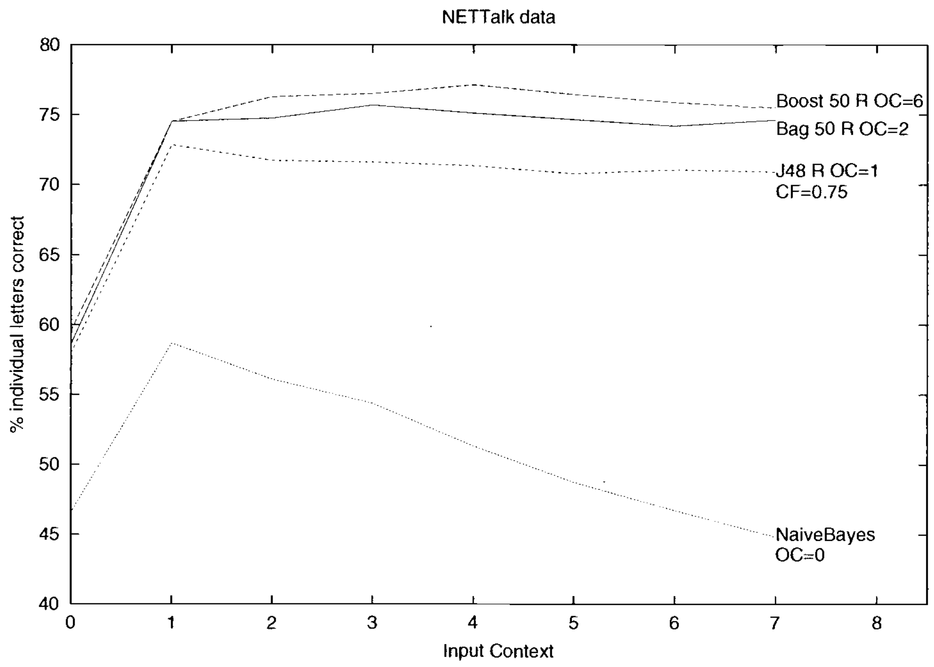

FIGURE 3.1: NETTalk: % of individual letters correct

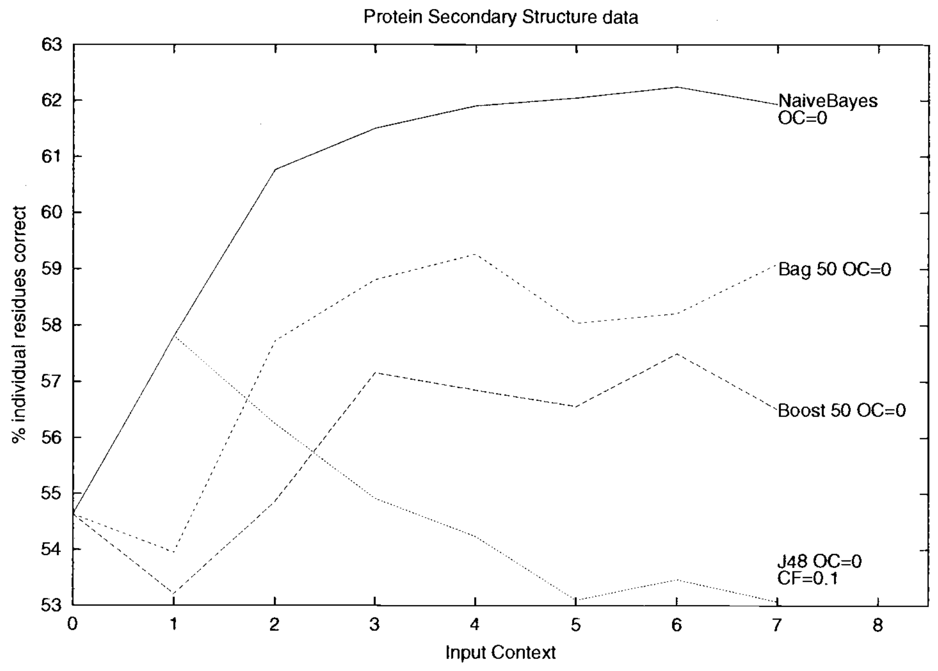

FIGURE 3.2: Protein Secondary Structure: % of individual residues (amino acids) correct

Figure 3.1 shows performance curves for the four algorithms applied to the NETtalk data. The performance criterion is the number of letters in the test dataset predicted correctly. Naive Bayes works very poorly on this problem. Peak performance is obtained with no output context and an input context of 2 which is a simple 5-letter window. A clear case of overfitting is exhibited as the input context increases.

The best configuration for a single decision tree is the following: input context of 1 (i.e., 3-letter window), output context of 1, and a pruning confidence factor of 0.75. Performance starts dropping slightly for greater input contexts. Bagging achieves significantly better performance, and it attains maximum per-

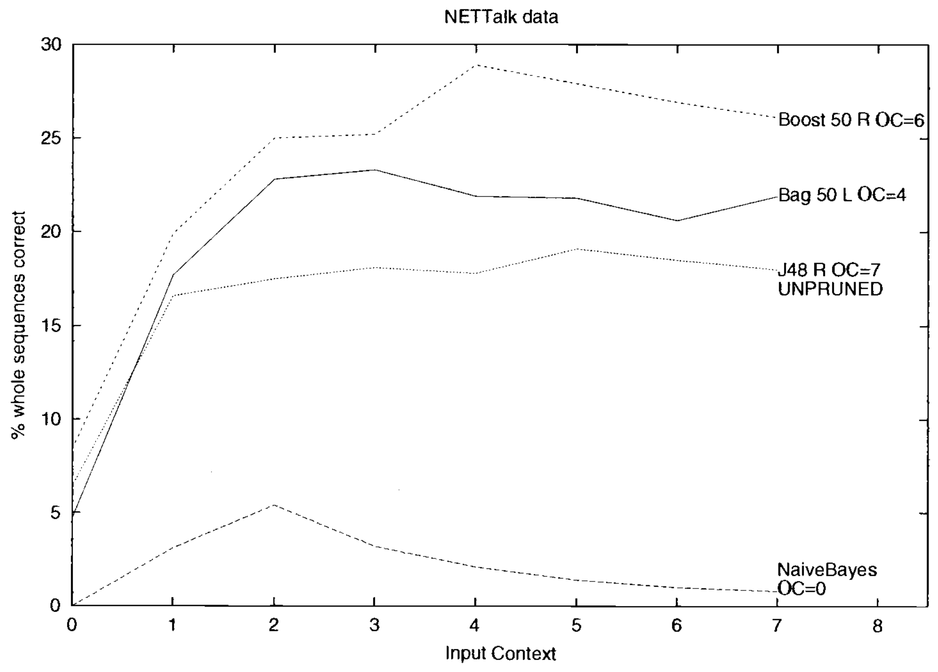

FIGURE 3.3: NETTalk: % of whole words correct

formance with an input context of 3, output context of 2 (and 50 bagged trees). Boosted decision trees do somewhat better than Bagging. They achieve maximum performance with an input context of 4 and an output context of 6. So there is a pattern: as we move from a single tree ($IC = 1, ROC = 1$) to Bagging ($IC = 2, ROC = 2$), to Boosting ($IC = 4, ROC = 6$), the learning algorithms are able to handle larger input and larger output context.

From this figure, we draw the following conclusions. First, the best configuration depends on the learning algorithm. Second, as expected, all of the algorithms exhibit some overfitting when the input context becomes too large. Third, all of the algorithms give very poor performance with an input context of 0 (i.e., only the one current letter in the window) which is equivalent to treating

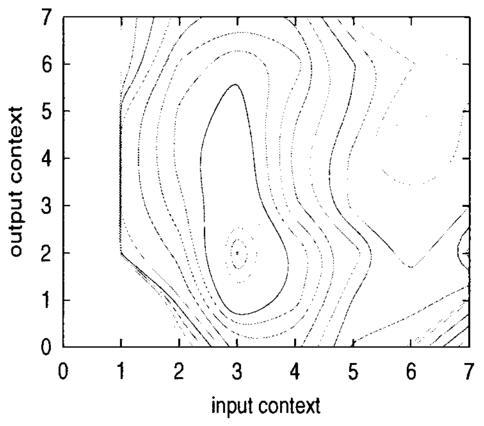

FIGURE 3.4: Contour plot showing Bagging performance as a function of input and output context.

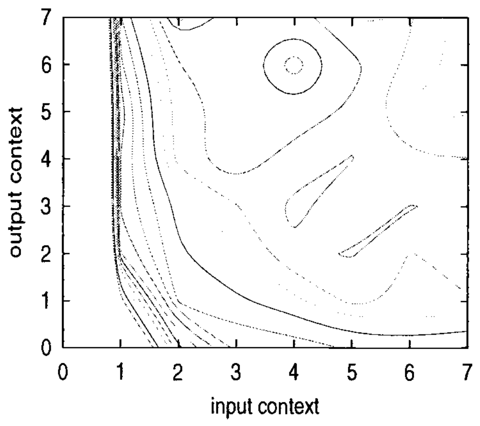

FIGURE 3.5: Contour plot showing boosting performance as a function of input and output context.

this problem as a standard supervised learning problem. Fourth, all algorithms obtain better results using right output context rather than left output context. This is consistent with the results of Bakiri (1991).

Figure 3.2 shows the performance of the four algorithms applied to the Protein dataset. In this domain, naive Bayes gives the best performance ($IC = 6$). The second best is bagged decision trees (IC=4). The third best is adaboosted decision trees ($IC = 6$), and the worst is a single decision tree with strong pruning ($IC = 1$, pruning confidence 0.1). Note that in all cases, an output context of 0 is preferred. Most state-of-the-art secondary structure prediction methods use non-recurrent sliding windows coupled with better input features and additional post-processing of the predictions (Jones, 1999).

Figure 3.3 shows the four algorithms applied to the NETtalk data again, but this time the evaluation criterion is the number of whole words in the test set predicted correctly. This graph is somewhat similar to the one in Figure 3.1, except that the peak performances now have different window sizes. Bagging does better with a smaller input context and a larger output context. Boosting uses the same output context of 6 but performs better with an input context of 4 instead of 3. A big surprise is that J48 gives its best performance with an output context of 7 (and no pruning). Even naive Bayes changes its best configuration from input context 1 to input context 2. The major conclusion to be drawn from this figure is that the optimal choice of input and output context depends on both the learning algorithm and the evaluation criterion.

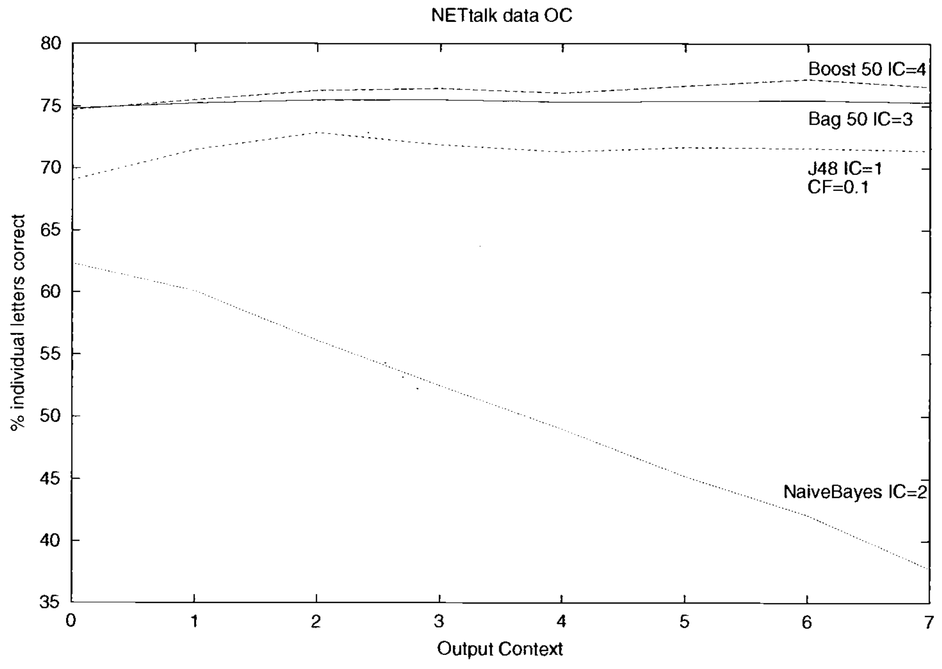

FIGURE 3.6: NETTalk: % of individual letters correct as a function of output context

Figures 3.4 and 3.5 show contour plots of the performance of Bagging and

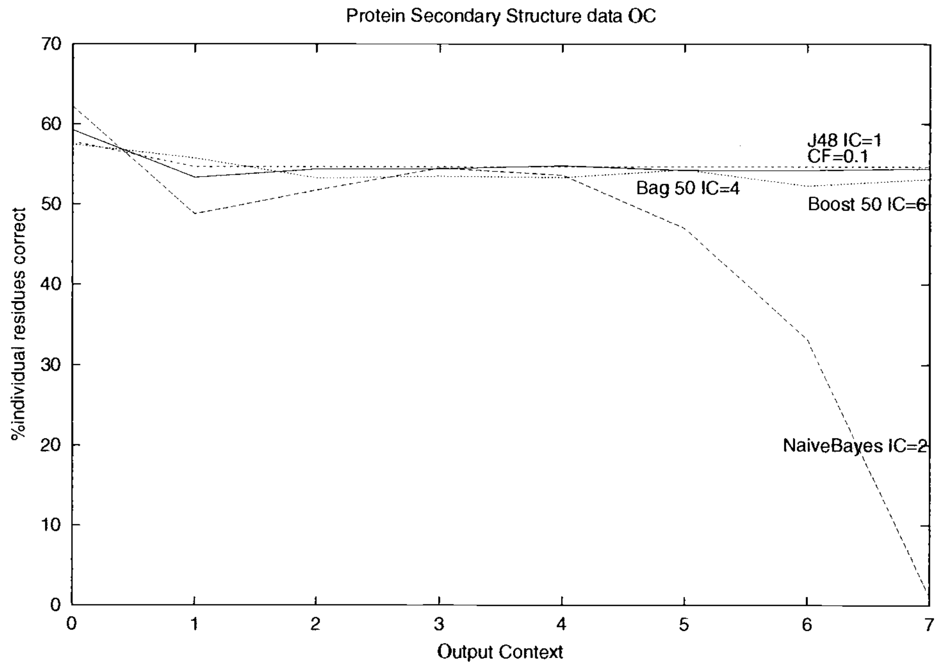

FIGURE 3.7: Protein Secondary Structure: % of individual residues (amino acids) correct as a function of output context

Boosting for various input and right output contexts. For Bagging, the maximum at input context 3, output context 2 is clearly visible as is the maximum at input context 4 and output context 6 for Boosting. There is some evidence for a trade-off between input and output context. For example, decreasing input context does allow us to increase output context or vice versa to a certain extent. This is the case, in Figure 3.5. This is less evident in Figure 3.4. However, there is a large plateau of good performance when the input and output contexts are sufficiently large. Here shrinking either input or output context hurts performance.

Figures 3.6 and 3.7 show the performance varying with output context for

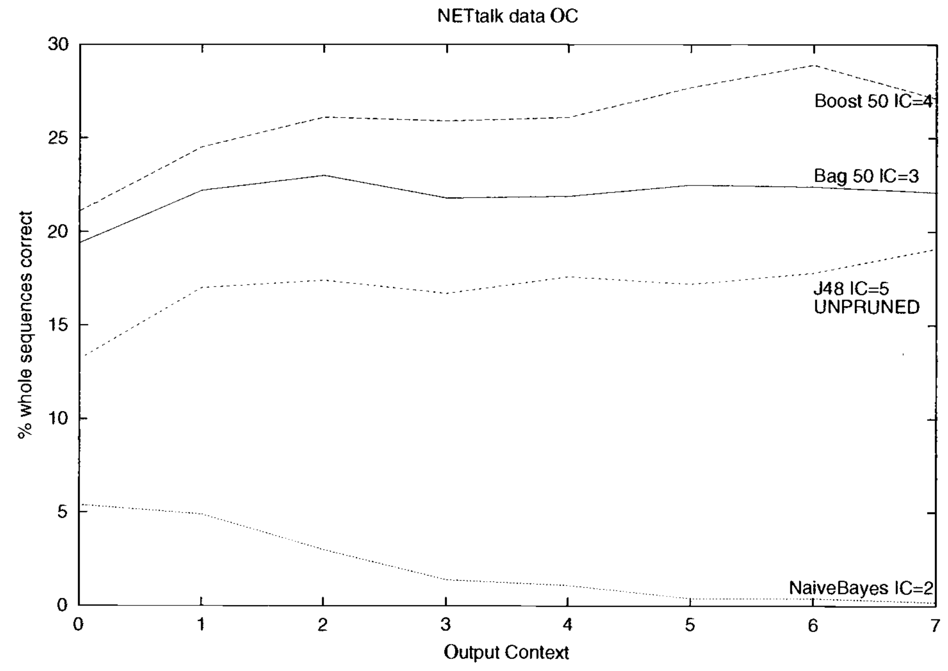

FIGURE 3.8: NETTalk: % of whole words correct as a function of output context

the different learning algorithms. The performance criterion is the number of individual examples predicted correctly. Here again all other parameters including the input context are chosen so as to maximize the performance. The input context for every curve is specified in a label adjacent to it.

We observe here that the curves for Bagging, Boosting and J48 are nearly flat, which shows that the amount of output context has little impact on their performance. In comparison, Figure 3.1 showed that the amount of input context has a much greater impact in the sense that it must be at least 1 in order to get a reasonable performance. However, for naive Bayes, increasing the output context has a devastating effect — performance drops rapidly and monotonically as the output context is enlarged.

The picture changes when we consider the percentage of whole sequences correctly classified as seen in Figure 3.8. Here, increasing the output context improves the accuracy of J48 and Boosting substantially, and it improves Bagging to a lesser extent. Again the accuracy of naive Bayes decreases with increasing output context. Clearly naive Bayes overfits for large input and output window sizes.

CHAPTER 4

DISCUSSION

It is clear from the results obtained that the optimal window size depends on a lot more than the underlying domain. In this chapter we attempt to explain why this is so. The problem domain certainly has information to provide, but it is not sufficient for calibrating a sequential data classifier. In order to explain the results we turned to the popular and widely accepted bias-variance theory. Bias-variance theory employs the standard statistical analysis that views each training set as a random sample from a population of possible training sets. It decomposes the prediction error of a classifier into three factors: (a) the bias, or systematic error of the learning algorithm, (b) the variance, or variation of the learning algorithm that results from random sampling of the given data set from the population, and (c) the noise, or variation in the class labels due to factors that cannot be predicted from the input features. In continuous prediction problems where the error is measured by the squared difference between the predicted and observed values of the response variable, the error can be decomposed into a sum of the squared bias, the variance, and the noise. In classification problems where the error is measured by the error rate, the decomposition is more complicated, and there has been considerable debate about the most appropriate bias-variance-noise decomposition. We will employ the decomposition developed by Domingos (2000). He developed his decomposition for the case where there are only two possible classes. We will first present his decomposition and then show how to extend it for more than two classes.

Consider that you havė lots of different training sets (of equal size) drawn from the same population at random. Consider that you have a test set that is representative of the population. Consider a classifier h trained on each of the training sets and tested on each example (x, y) of the test set. Let y_m be the majority prediction among all the classifiers for (x, y). Also consider that the true function we are trying to learn is f. There are 3 components that determine whether the prediction $y = h(x)$:

Noise: $y = f(x)$? (Is the label on x correct?)

Bias: $f(x) = y_m$? (Is the majority prediction among all classifiers correct?)

Variance: $y_m = h(x)$? (Does the majority prediction y_m match the prediction of the particular hypothesis h?)

To simulate this analysis using real datasets, we generate 200 bootstrap replicates of the training data (X, Y). On each bootstrap replicate, we train a classifier h using a learning algorithm that attempts to learn the true function f. Then, we classify each test point x on each classifier. We assume that the noise is zero.

Domingos developed a case analysis of error. If there is no noise, this analysis can be shown as in Figure 4.1 The first level in this tree structure checks for bias, and the second level checks for variance. In the case where the classification is not biased, if there is no variance, then the prediction is correct. Otherwise variance causes error. This type of variance is called unbiased variance and it is undesirable. In the case where the classification is biased, lack of variance is bad (biased variance in this case), because biased variance will cause the prediction to differ from the biased classification. In the 2-class case, if the prediction if not equal to the wrong class, it has to be equal to the right class. Therefore

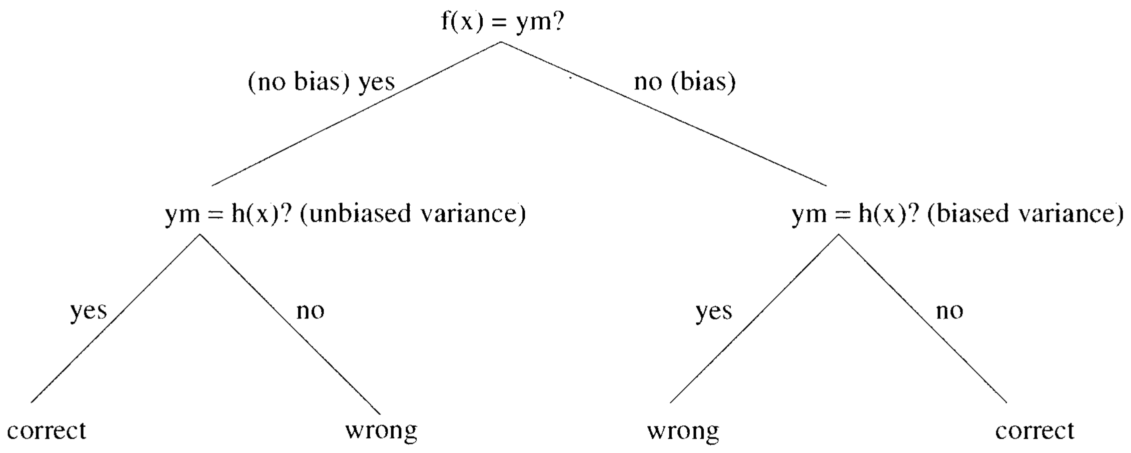

FIGURE 4.1: A Case Analysis of Error for 2 classes

biased variance is desirable variance. Over the entire test set, the total bias, unbiased and biased variance is calculated by simple summation and the error is measured as follows:

$$Error(l) = \sum Bias(b) - \sum biasedVariance(V_b) + \sum unbiasedVariance(V_u)$$

In order to perform a bias variance analysis of our results, we extended Domingos' work for the case where there are multiple classes. A case analysis of error for the multiple class case can be shown as in Figure 4.2. Once again we assume there is no noise. The only difference from the 2-class setting is for the biased variance case. In this case, the classification is biased and suffers from variance as well. But this biased variance is not necessarily good, because with more than 2 classes now, it is not enough for the prediction to be unequal to a wrong answer. Since there are multiple wrong answers here, the prediction can be biased, differ from the majority prediction, and still be wrong. This brings us to another kind of variance we term "biased unlucky variance". It is unlucky, because even when the classification decision facing this kind of variance differs

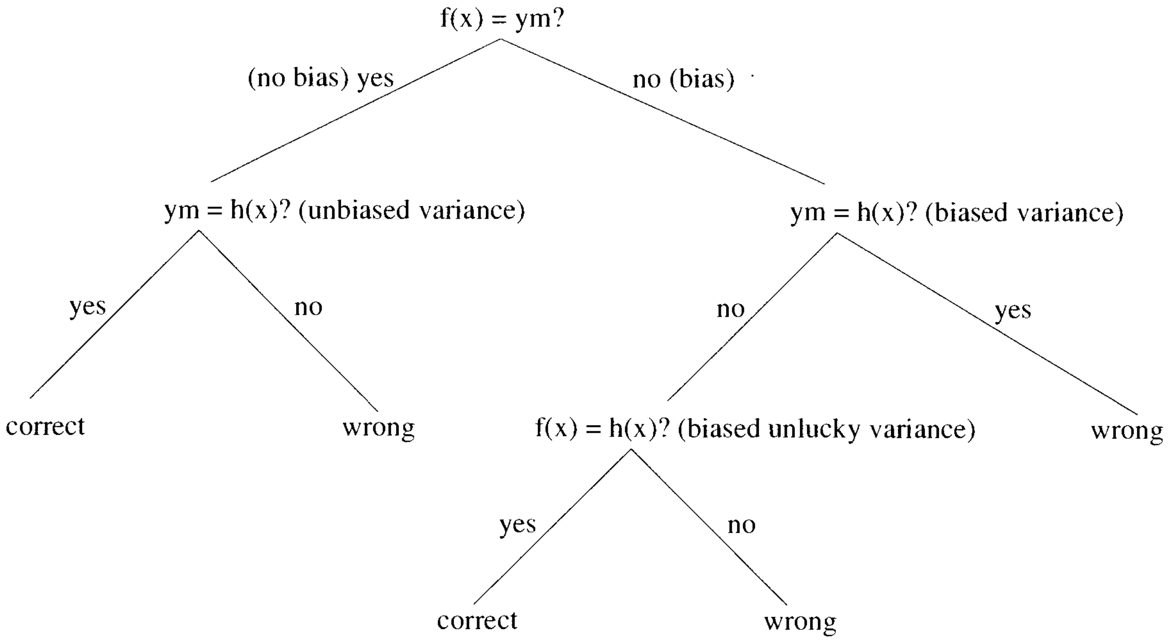
f(x) = ym?
(no bias) yes
no (bias)
ym = h(x)? (unbiased variance)
ym = h(x)? (biased variance)
yes
no
no
yes
correct
wrong
f(x) = h(x)? (biased unlucky variance)
wrong
yes
no
correct
wrong

FIGURE 4.2: A Case Analysis of Error for multiple classes

from the incorrect majority, it is wrong and hence loses on both fronts. The error decomposition now has a new term and can be written as follows:

$Error(l) = \sum Bias(b) - \sum biasedVariance(V_b) + \sum unbiasedVariance(V_u) + \sum BiasedUnluckyVariance(V_bu)$

Figures 4.3 and 4.4 show the results of our experiments on the bias-variance analysis of single decision trees and 50 bagged trees respectively. For the purpose of these experiments, we have chosen the NETtalk data and assumed there is no noise in the data. The Figures 4.3 and 4.4 plot bias, variance, and error rate against output context for no input context. It can be seen from both these curves that bias decreases with increasing context (window size) and net variance increases with increasing context (window size). This corresponds to our intuition. Suppose the data was generated by a second order hidden Markov

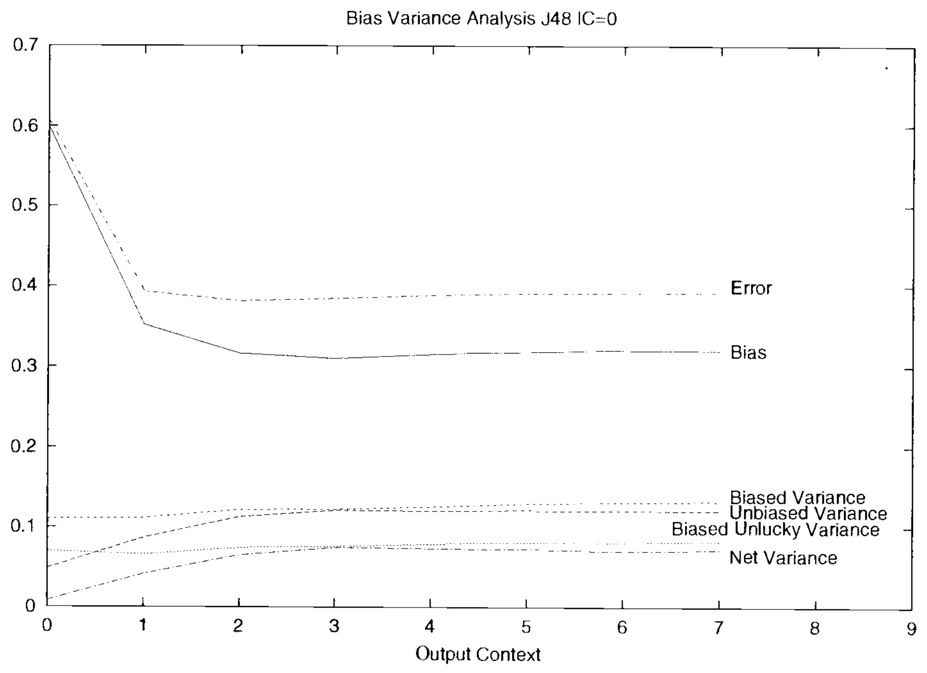

FIGURE 4.3: Bias Variance Analysis of J48 against output window size with IC=0 for NETtalk data

model. If a learnt classifier now used an output context of 0 or 1, there will be a bias, because the learnt classifier cannot represent the true function. The greater the window size, greater is the probability that the true order or the generative model is encompassed by the learnt classifier. Hence there is a lower bias. At the same time, a large window equates to a large number of features in the sliding window realm. Variance increases with the number of features in the data set and hence should also increase with increasing window sizes.

Figures 4.5, 4.6, and 4.7 compare the error rate, bias, and net variance curves of J48 decision trees and 50 bagged trees for respectively 3 different output contexts 0, 4, and 7. Each of the plots tell the same story. Bagging J48

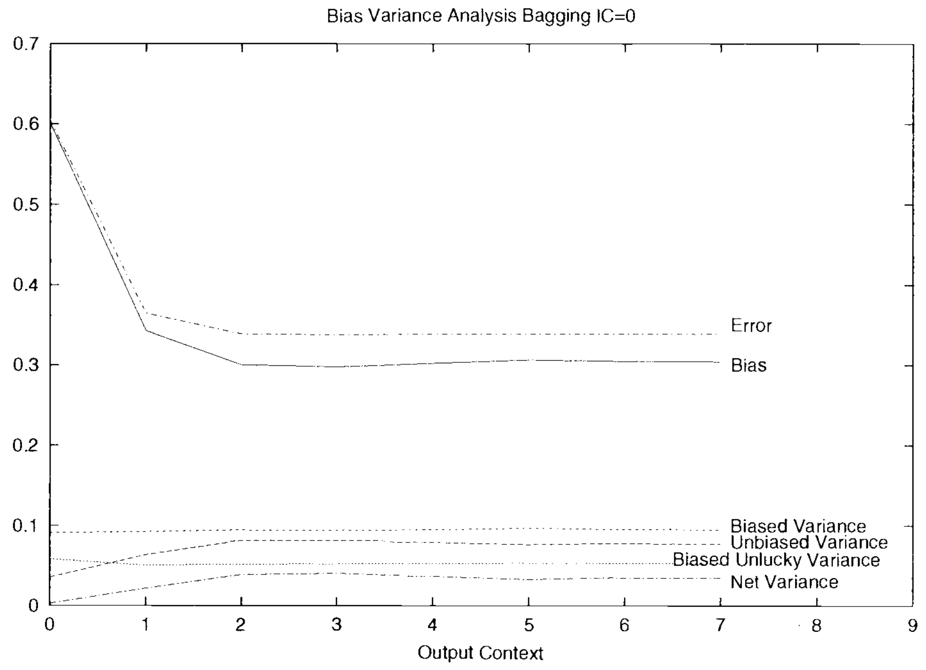

FIGURE 4.4: Bias Variance Analysis of Bagged J48 against output window size with IC=0 for NETtalk data

decision trees gives a reduction in net variance as it should according to theory. The apparent small reduction in bias with Bagging is actually because of the fact that to optimize performance the bagged trees have been left unpruned whereas single decision trees have been pruned using pessimistic pruning with a confidence factor of 0.25. Prunning reduces variance but increases bias in trees. With Bagging however, the aggregation reduces variance, so better performance is obtained by growing the trees to full depth and thereby reducing bias. The bias curves for Bagging and J48 are almost overlapping indicating there is not much difference in the bias for all window sizes. Bias decreases steadily as the window size increases. It decreases up to an input context of 3 and then remains

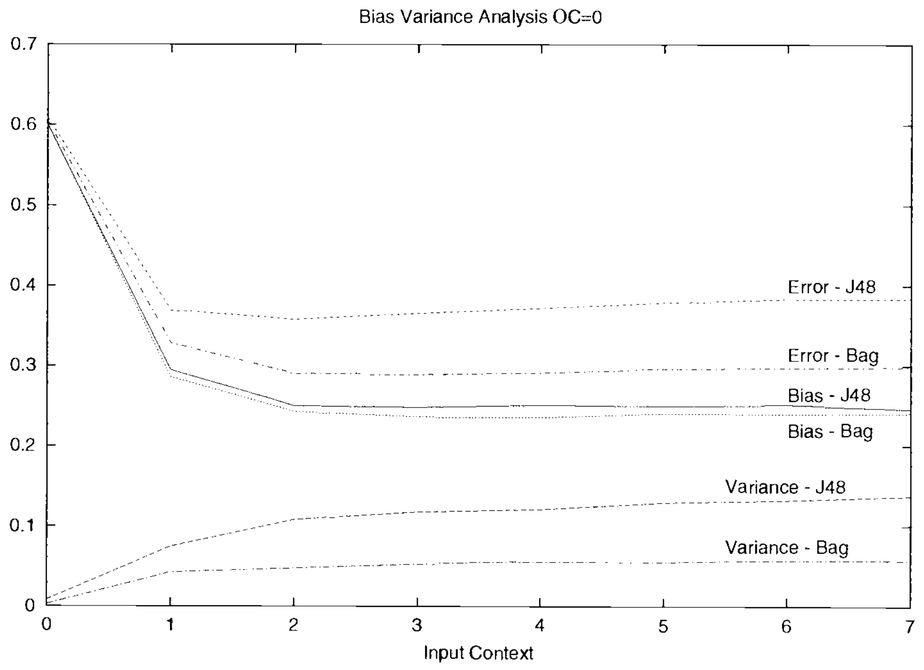

FIGURE 4.5: Bias Variance Analysis of J48 and Bagging as a function of input window size with OC=0 for NETtalk data

more or less constant. The variance curves are a lot different. There is a huge gap in variance between J48 and Bagging, and this difference is increasing with increasing window size. Clearly Bagging is able to eliminate the variance caused by using single decision trees. Variance due to bagged decision trees increases much slower than variance due to J48. The optimal operating point (the point where error is minimum) is at an input context of 2 for J48 and an input context of 3 for Bagging in Figure 4.5. Bagging, therefore, employs a larger window and obtains a better performance. Similarly in Figures 4.6 and 4.7, the window size at which Bagging reaches its optimal point is greater than the window size at which J48 reaches its optimal point.

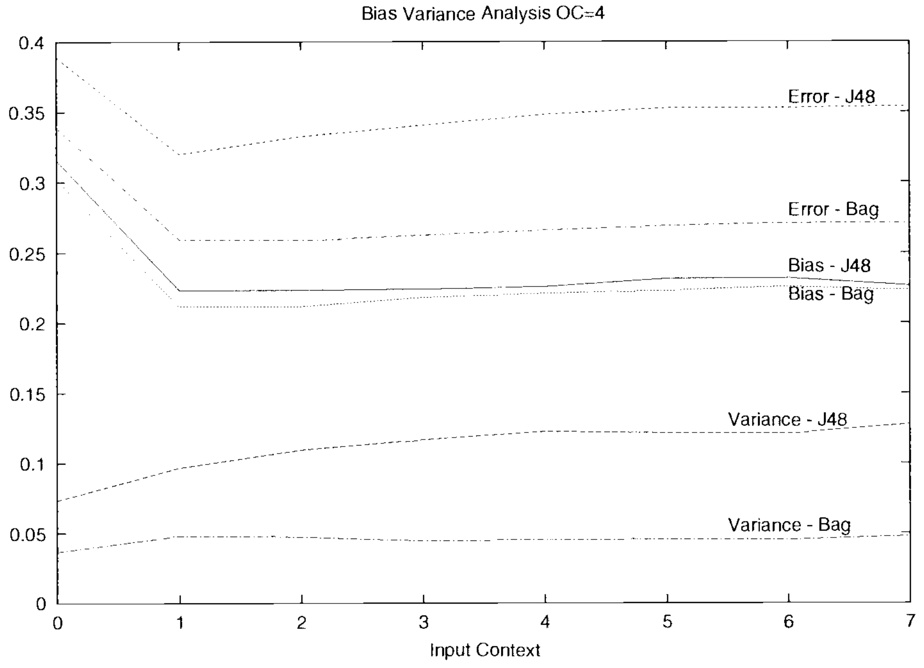

FIGURE 4.6: Bias Variance Analysis of J48 and Bagging as a function of input window size with OC=5 for NETtalk data

This is the point where the story gets interesting. With standard supervised learning, sources of bias and variance for the classifier are the training data and the learning algorithm which generates the classifier. However with sequential supervised learning, as we saw earlier, the window size is an additional source of bias and variance. A bagged classifier can afford a larger window size, because Bagging has lower variance. The extra variance generated by a greater window size is eliminated by bagging the classifier. The bagged classifier then uses this larger window size to obtain some additional bias reduction. This is why Bagging can use a larger window size and yet achieve better performance. The same argument can be extended to Boosting, which is a bias and variance

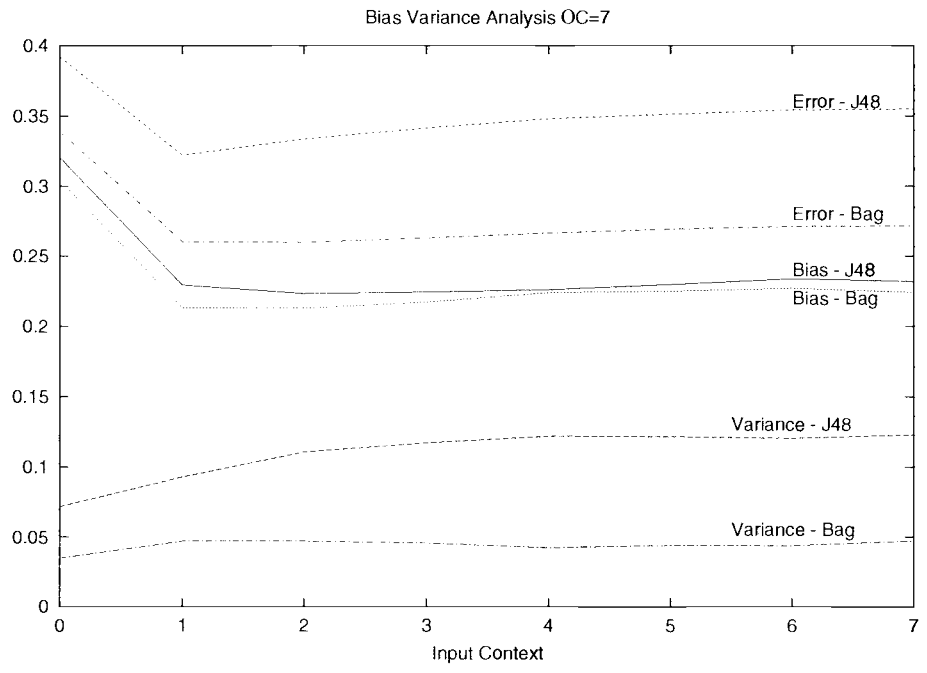

FIGURE 4.7: Bias Variance Analysis of J48 and Bagging as a function of input window size with OC=7 for NETtalk data

reduction technique. This confirms the idea that differences in optimal window sizes for different learning algorithms are due to the different bias variance profiles of these algorithms.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

All methods for solving sequential supervised learning problems must confront the question of how much context to employ. For HMMs and conditional random fields, the question of output context becomes the question of the order of the Markov process (first order, second order, etc.). HMMs cannot handle large input contexts (at least not without violating the assumptions of the generative model), but CRFs can. Hence, CRFs also must face the choice of the size of the input context. The choice of input and output context is analogous to classical feature selection. One common approach to feature selection is to fit some simple model to the data and/or compute some figure of merit for the informativeness of each attribute (Kira & Rendell, 1992; Koller & Sahami, 1996). The experiments reported here show that this will not work for SSL problems, because the correct choice of input and output contexts depends on the learning algorithm. Although a simple bias-variance analysis explains the results, the input and output contexts chosen by some simple method (e.g., naive Bayes) are not good choices for boosted decision trees. A consequence of this is that the input and output contexts need to be chosen by cross-validation or holdout methods.

The experiments have also shown that the optimal input and output contexts depend on the performance criterion. Window sizes that maximize the correct classifications of individual examples are not the ones that maximize the correct classifications of whole sequences and vice versa.

An important question for future research is to understand why different window sizes are required for different performance criteria (whole sequences vurses individual items). Our bias-variance analysis explains why Bagging and Boosting are able to handle large window sizes, but it does not explain why larger output contexts are more valuable for whole sequence classification. Intuitively, whole sequence classification requires greater integration of information along the sequence. The primary way of integrating information along the sequence is through the output context. But this intuition needs to be made more precise in order to obtain a complete theory of sequential supervised learning.

BIBLIOGRAPHY

Bakiri, G. (1991). Converting English text to speech: A machine learning approach. Tech. rep. 91-30-2, Department of Computer Science, Oregon State University, Corvallis, OR.

Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T., Nigam, K., & Slattery, S. (1998). Learning to extract symbolic knowledge from the World Wide Web. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pp. 509–516 Menlo Park, CA. AAAI Press/MIT Press.

Jones, D. T. (1999). Protein secondary structure prediction based on position-specific scoring matrices. *Journal of Molecular Biology*, *120*, 97–120.

Kira, K., & Rendell, L. A. (1992). A practical approach to feature selection. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 249–256 San Francisco, CA. Morgan Kauffman.

Koller, D., & Sahami, M. (1996). Toward optimal feature selection. In *International Conference on Machine Learning*, pp. 284–292.

Lafferty, J., McCallum, A., & Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning*, pp. 282–289 San Francisco, CA. Morgan Kaufmann.

Màrquez, L., Padró, L., & Rodríguez, H. (2000). A machine learning approach to POS tagging. *Machine Learning*, *39*(1), 59–91.

Qian, N., & Sejnowski, T. J. (1988). Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, *202*, 865–884.

Quinlan, J. R. (1993). C4.5: Programs for Empirical Learning. Morgan Kaufmann, San Francisco, CA.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257–286.

Sejnowski, T. J., & Rosenberg, C. R. (1987). Parallel networks that learn to pronounce english text. *Journal of Complex Systems*, *1*(1), 145–168.

McCallum, A., Freitag, D. & Pereira, F. (2000). Maximum Entropy Markov Models for Information Extraction and Segmentation. In *Proceedings of the 17th International Conference on Machine Learning*, pp. 591–598 Stanford, CA. Morgan Kaufmann.

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

APPENDIX

APPENDIX

RSW DOCUMENTATION

A.1 WEKA structure

A.1.1 Overall Structure

The WEKA system is a Java-based package. Its developers at the University of Waikato NZ have systematically divided this package into a series of supplementary and consistent sub-packages. Each sub-package contains sub-sub-packages and so on finally terminating in class files. Each class file has a specific purpose and hence a very targeted implementation. Every sub-package is generated according to functionality and code for various algorithms is grouped with the same concept. WEKA always has more than one version available. A stable version is available for the general users. For developers there are development versions. Both these versions have a graphical user interface. A stable book version, which is a subset of the GUI version is also available. There are 10 packages that together make the WEKA system. These are as follows:

1. weka.associations: As the name suggests, the classes in this package are for generating object instances that apply association rules to data and learn association rules from data.

2. weka.attributeSelection: The purpose for the classes in this package is very clear from the name. These classes are largely used to manipulate the

feature space to be used by various classifiers.

3. weka.classifiers: These classes implement various learning algorithms that generate classifiers. This is a very elaborate package, and we will discuss it in some more detail later.

4. weka.clusterers: These classes implement clustering algorithms.

5. weka.core: This package contains all basic functionality for working with data sets. It also includes classes that construct datatypes useful for manipulation by classifiers, filters and so on.

6. weka.datagenerators: Instantiations of classes in here are used for generation of data according to some pre-defined format.

7. weka.estimators: Estimators give information about some parameter of an underlying distribution. The classes in this package are implement such algorithms.

8. weka.experiment: This package provides extra functionality for doing structured experiments using WEKA.

9. weka.filters: Filters are instances of classes that manipulate the training and test datasets. Manipulation can take the form of masking attributes, choosing subsets of training examples based on some criteria, converting categorical feature values to binary and so on. Filters largely use the datatypes provided by the core package, and they are generally used prior to passing the dataset on to classifiers for training or evaluation. We will discuss this package in some more detail later.

10. weka.gui: This package is responsible for the graphics while intense code is running in the background. It obviates the need to check references and manuals to look up command line options.

Of these 10 top level packages that make up WEKA, the weka.classifiers and weka.filters have been extended by the work described in this thesis. The weka.filters package is divided into two sub-packages namely the attribute filters and the instance filters. As the names suggest, attribute filters are those that manipulate attributes, and instance filters are those that manipulate instances.

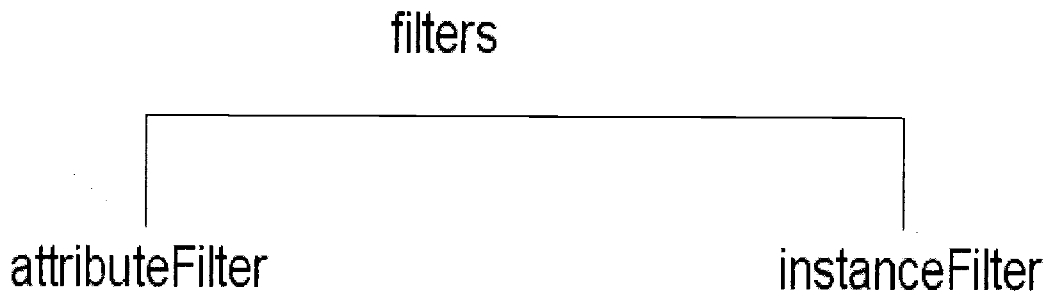

FIGURE .1: The filters sub-package

The main template class "Filter" deserves some detailed explanation in the form of documentation here.

A.1.2 filters

Class Name: Filter

Parent Class: java.lang.Object

General description:

This class is abstract, which means that it is not meant for instantiation but only to serve as a template to build implementations in the category of filters. As explained earlier, filters are manipulators of datasets. However, they do not always remove something from the dataset as the name might seem to suggest. A NominalToBinaryFilter, for example, replaces nominal (categorical or enumerated) attributes with a number of binary attributes. There are many subclasses to this class. Filter provides the following member variables and methods that its subclasses can build on for thier specific algorithms.

Relevant Member Variables:

1. m_InputFormat: Since filters alter the format of the dataset (number of attributes, instances etc), objects are required to store the source and target formats. This variable is the store for the source format. The datatype is "Instances".

2. m_OutputFormat: Store for the target format. The datatype is "Instances".

Relevent Member Functions:

1. setInputFormat:

   Input: data points (examples) in a datatype object "Instances".

   Output: None.

   Description: This method accepts the source data points and stores them into m_InputFormat. It also calls the setOutputFormat method from within itself if required.

2. setOutputFormat:

   Input: data points (examples) in a datatype object "Instances".

   Output: None.

   Description: This method is responsible for creating the target format into which the given format of instances is to be converted.

3. useFilter:

   Input: Data points in a datatype object "Instances" and the filter object.

   Output: Modified data points in a datatype object "Instances".

   Description: This method is probably the most important method for filters. It is a static method (meaning that it relates to the whole class rather than each instantiation), and it works as the interface between the calling code and the filter implementation.

4. batchFinished:

   Input: None.

   Output: true or false.

   Description: This method is where the filter algorithm is implemented. Every instance in the source dataset is converted to its target format. The boolean output is just an indication of whether the process was carried out successfully.

An example of filter use:

Instance source = ... obtained from somewhere.

```
SomeFilter theFilter = new SomeFilter();
theFilter.setInputFormat(source);
source = Filter.useFilter(source, theFilter);
```

A.1.3 classifiers

The package weka.classifiers is huge and understandably so. Directly under this package there are a few class implementations among which those of chief interest to us are the Classifier.class (a template for construction of all data classifiers), the Evaluation.class (an instantiation of this class provides functionality for generating statistical information when some test data is classified), and the DistributionClassifier.class (a template for classifiers that output a probability distribution over the possible class values).

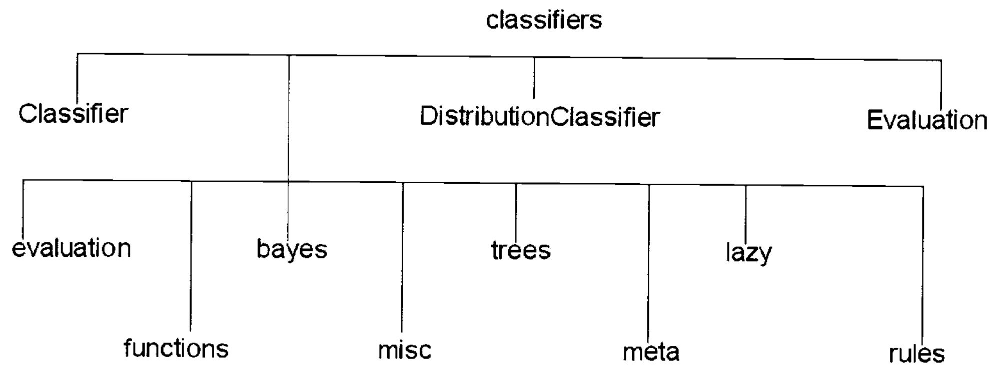

FIGURE .2: The classifiers sub-package

The classifiers package is further divided into 8 sub-packages. Each contains classes that represent implementations of classifiers that can be placed into a certain category according to a set of criteria. For example, Bagging and Boosting are classification algorithms that use an ensemble of base classifiers and are therefore placed under the subpackage "meta". All decision tree algorithms come under the subpackage "trees". All Bayesian classifiers come under "bayes" and so on. Again the main template class "Classifier" deserves detailed explanation before we move on.

Class Name: Classifier

Parent Class: java.lang.Object

General description:

This class is abstract, which means that it is not meant for instantiation but only to serve as template to build implementations in the category of classifiers. There are many known subclasses to this class and more are coming up. Classifer provides the following member variables and methods that its subclasses can build on for their specific algorithms.

Relevant Member Variables: None

Relevent Member Functions:

1. buildClassifier:

   Input: training examples in a datatype object "Instances".

   Output: None.

   Description: This method is the implementation of the learning algorithm that the class represents. The subclass of Classifier uses the training instances supplied to generate the classifier. It is only after the BuildClassifier method is called with appropriate training examples that classifications can be made on unseen data.

2. classifyInstance:

   Input: One test example in a datatype object "Instance".

   Output: Classification decision as a number (double) corresponding to a class.

   Description: This method is where the subclass of Classifer supports implementation of the classification algorithm. It only classifies a single example. To classify a number of test examples, this method is called many times (e.g., inside a loop).

An example of classifer use:

Instances train = ... obtained from some source.

Instance test = ... obtained from some source.

SomeClassifier classifier = new SomeClassifier();

classifier.buildClassifier(train);

double result = classifier.classifyInstance(test);

A.2 The RSW package

Our system with the main class RSW is not a separate package in itself. There are 5 classes other than RSW that aid in the working and presentation of a complete sequential data classifier. In all, our system consists of the following classes:

1. SequentialClassifier
2. DistributionSequentialClassifier
3. RSW
4. Windowise
5. SequentialEvaluation
6. SeqCVParameterSelection

Let us look at each one in detail.

A.2.1 SequentialClassifier

Class Name: SequentialClassifier

Parent Class: weka.classifiers.Classifier

General Description: This class is abstract, and it serves as a template for sequential data classifiers including RSW and any other potential classifiers tuned specifically for classification of sequential data.

Relevant Member Variables: None.

Relevant Member Functions:

1. classifySequence:

   Input: Data points in a datatype object "Instances".

   Output: An array of classifications in the form of numbers (double) corresponding to a probability distribution over the space of class labels.

   Description: This function is the sequential counterpart of the method "classifyInstance" in Classifier. Input is taken as an "Instances" instead of "Instance", because classification decisions cannot be made unless the sequence in which a particular data point occurs is known completely. This class has been placed inside the weka.classifiers package.

A.2.2 DistributionSequentialClassifier

Class Name: DistributionSequentialClassifier

Parent Class: weka.classifiers.SequentialClassifier

General Description: This class is abstract, and it serves as a template for sequential data classifiers that return a probability distribution over possible class values during classification instead of just a single value.

Relevant Member Variables: None.

Relevant Member Functions:

1. distributionForSequence:

   Input: Data points in a datatype object "Instances".

   Output: An array of arrays each representing a probability distribution.

   Description: This function is the sequential counterpart of the method "distributionForInstance" in the class DistributionClassifier. "distributionForInstance", as the name suggests, returns a distribution over the space of posiible class values for an example. Once again the input is taken as an "Instances" object instead of an "Instance" object, because classification decisions cannot be made unless the sequence in which a particular data point occurs is known completely. This class has been placed inside the weka.classifiers package.

A.2.3 RSW

Class Name: RSW

Parent Class: weka.classifiers.DistributionSequentialClassifier

General Description: This is the heart of our system. It is a class that directly descends from DistributionSequentialClassifier and provides implementations for all its methods, mainly "buildClassifier" and "distributionForSequence". The sliding window classification algorithm explained in the introduction is implemented here. "windowisation" of the data performed by the Windowise filter is the soul of this algorithm. "windowisation" is done at the time of training and testing both. This class has been placed under the weka.classifiers.meta package, because it conforms to that category. Like Bagging and Boosting, it uses a base classifier and builds functionality on it.

A.2.4 Windowise

Class Name: Windowise

Parent Class: weka.filters

General Description: The process of windowisation was explained in the introduction chapter of this thesis. This filter takes as input four numbers specifying the sizes of the input and output contexts required for generating the sliding window: *LIC* (the amount of left input context), *RIC* (the amount of right input context), *LOC* (the amount of left output context) and the *ROC* (the amount of right output context). For RSW, one of *LOC* and *ROC* must be zero. Due to design constraints in the WEKA package, this filter cannot be used by itself. It has to be used through another program. In other words, the user cannot run this code from the command line, feed it a dataset, and get a windowised data set as output. However this is not a fundamental limitation, and it might be sorted out in future versions. Since the windowising technique affects both the attributes (features) and the instances (examples) in a dataset, the Windowise class has been placed under the global filters package itself.

A.2.5 SequentialEvaluation

Class Name: SequentialEvaluation.

Parent Class: java.lang.object.

General Description: Classifiers in WEKA use a class called "Evaluation". This class is responsible for generating statistics on the tasks the user wants to perform, like the listing the number of test examples correctly classified, the confusion matrix and so on. Taking inspiration from there, we wrote an evaluation class "SequentialEvaluation" that does the same for sequential data classifiers. The following is a sample output from running Naive Bayes on a protein secondary structure classification dataset through SequentialEvaluation.

```
weka.classifiers.meta.RSW

Time taken to build model: 3.8 seconds

Time taken to test model on training data: 4.07 seconds


=== Error on training data ===

Correctly Classified Instances        9643           53.2615 %

Incorrectly Classified Instances      8462           46.7385 %

Correctly Classified Sequences           0            0      %
```

```
Incorrectly Classified Sequences      111              100       %
Kappa statistic                        -0.0081
Mean absolute error                     0.2334
Root mean squared error                 0.4575
Relative absolute error                78.049 %
Root relative squared error           118.3251 %
Total Number of Instances           18105


=== Confusion Matrix ===

    a     b    c     d   <-- classified as
 9607   138    0   123     a = _
 3599    36    0     1     b = e
 4519    77    0     5     c = h
    0     0    0     0     d = #



=== Error on test data ===

Correctly Classified Instances       1903             54.0625 %
Incorrectly Classified Instances     1617             45.9375 %
Correctly Classified Sequences          0              0       %
Incorrectly Classified Sequences       17            100       %
Kappa statistic                        -0.0056
Mean absolute error                     0.2297
Root mean squared error                 0.4537
Relative absolute error                76.7851 %
```

```
Root relative squared error            117.301  %
Total Number of Instances             3520


=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 1903    0    0   20    a = _
  748    0    0    0    b = e
  849    0    0    0    c = h
    0    0    0    0    d = #
```

A.2.6 SeqCVParameterSelection

Class Name: SeqCVParameterSelection

Parent Class: SequentialClassifier

General Description: In our experience, it is important to carefully choose the size of the input and output contexts to obtain good results. If the contexts are too large, performance is damaged because of high variance (overfitting) by the base learning algorithm. If the contexts are too small, performance may be poor because not enough contextual information is available to the base learning algorithm. We have observed cases where the input context should be very small yet the output context is large and also cases where the output context should be zero and the input context should be large. Furthermore, the choice of context may also depend on whether you seek to maximize the number of elements correctly classified or the number of entire sequences correctly classified. A good way to choose the context parameter values (*LIC*, *LOC*, *RIC*, *ROC*) is to perform an internal cross-validation on the training data. RSW supports this with the SeqCVParameterSelection class. The SeqCVParameterSelection can be used from the command line exactly as you would use CVParameterSelection but with RSW as the base classifier. Please note that SeqCVParameterSelection will not accept any other base classifier.

A.3 Structure of the ARFF file for sequential data

In WEKA, input files containing datasets must be in a particular format called the ARFF format, in WEKA. For RSW, the ARFF file contains one example for each element of each training sequence. The data file has the standard ARFF format with the following extensions:

The first attribute in each line must be the sequence number. Each training example (x, y) is assigned a unique sequence number (in ascending order counting from 1). The second attribute in each line must be the element number. Each position (x_t, y_t) for $t = 1, \dots, T$ (where T is the length of X and Y) must be assigned an element number (in ascending order counting from 1). The remaining attributes in each line provide features that describe x_t and the class label y_t. The following example ARFF file shows two training sequences (bad, BAD) and (feed, FEED). Note that each attribute and the class variable must specify an extra null value that is used to pad context that extends beyond the end of the sequence. This null value appears as the final value in the value list. In this case, we have specified "_" as the null value for both feature1 and class.

```
@relation SampleSequentialData

@attribute sequence_number numeric

@attribute element_number numeric
```

```
@attribute feature1 {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,
w,x,y,z,_}
@attribute class {A1, B1, C1, D1, E1, F1, _}
@data
1,  1,  b,  B1
1,  2,  a,  A1
1,  3,  d,  D1
2,  1,  f,  F1
2,  2,  e,  E1
2,  3,  e,  E1
2,  4,  d,  D1
```

The type for the sequence_number and element_number attributes must be numeric. The sequence numbers must start with 1 and proceed serially without any breaks or jumps in increasing order. The element numbers inside each sequence also must start at 1 and proceed serially in increasing order until the end of the sequence. The class attribute must be nominal. RSW converts the data structured as above into the windowised data as shown in the previous section. The non-nominal attributes remain unmodified.

A.4 Using RSW from the command line

RSW is a meta classifier (like Bagging and AdaBoostM1). Therefore, it requires a base classifier. RSW takes all options taken by any other classifier e.g., -t "train.arff" -T "test.arff" and so on. In addition to these, RSW requires 5 mandatory options.

-A size of the left input context for sliding window

-B size of the right input context for sliding window

-Y size of the left output context for sliding window

-Z size of the right output context for sliding window

For running classifiers and other code in general through WEKA, please see the WEKA Documentation. For documentation, versions, contribution and any other information on WEKA, visit
`http://www.cs.waikato.ac.nz/ml/weka/index.html`