An Immune System Approach to Document Classification

Jamie Twycross
Information Infrastructure Laboratory
HP Laboratories Bristol
HPL-2002-288
October 23rd, 2002*

E-mail: jamie@milieu3.net

artificial immune system, concept learning, classifier, machine learning, information retrieval, cooperative coevolution, feature extraction

The human immune system as a biological complex adaptive system has recently provided inspiration for a range of innovative problem solving techniques in areas such as computer security, knowledge management and information retrieval. In this dissertation the construction and performance of a novel immune-based learning algorithm is explored whose distributed, dynamic and adaptive nature offers many potential advantages over more traditional models. Through a process of cooperative coevolution a classifier is generated which consists of a set of detectors whose local dynamics enable the system as a whole to group positive and negative examples of a concept. The immune-based learning algorithm is tested in a rigorous and systematic manner, first on a standard classification problem and then, combined with an HTML feature extractor, on a web-based document classification task in the context of a system which allows users to perform document-based searches and automatically refine search results. The immune-based classifier is found to outperform traditional classification paradigms on both tasks. Further applications in community knowledge management systems, content filtering, recommendation systems and user profile generation are also directly relevant to the work presented.

* Internal Accession Date Only
Approved for External Publication
© Copyright Hewlett-Packard Company 2002

An Immune System Approach to Document Classification

Master's Thesis

Jamie Twycross[1]
COGS, University of Sussex, U.K.
Hewlett-Packard Research Labs, Bristol, U.K.

August 30, 2002

[1]jamie@milieu3.net

Acknowledgements

As an academic and creative work this thesis would not have been possible without the input and hard work of many people. Thanks to my colleagues at COGS and HP Labs Bristol, especially to Steve Cayzer, Inman Harvey and Dave Cliff. Thanks also to Mitch Potter and Paul White. In the broader context, none of this would ever have happened without the love and support of my friends, family, and, most of all, of Lorraine.

Contents

1 Introduction **1**

2 Background **6**
2.1 Information retrieval . . . 6
2.1.1 Feature extraction . . . 7
2.1.2 Concept learning . . . 8
2.1.3 Applications . . . 9
2.2 The human immune system . . . 11
2.3 Artificial immune systems . . . 15
2.3.1 General principles . . . 15
2.3.2 Applications . . . 16
2.3.2.1 Machine learning . . . 16
2.3.2.2 Anomaly detection . . . 18
2.3.2.3 Optimisation . . . 19
2.4 Evolutionary algorithms . . . 20
2.4.1 Cooperative coevolution . . . 20
2.4.2 Evolving concept learners . . . 21
2.4.3 Immune-based evolutionary algorithms . . . 22

3 Materials and methods **23**
3.1 The classifiers . . . 23
3.1.1 The artificial immune system classifier . . . 23
3.1.2 The naive Bayesian classifier . . . 24
3.2 The evolutionary algorithm . . . 24
3.2.1 The encoding scheme . . . 25
3.2.2 The fitness evaluation scheme . . . 28
3.3 Test data . . . 28
3.3.1 The voting problem . . . 28
3.3.2 The HTML document classification problem . . . 28
3.4 The feature extractor . . . 30
3.5 Summary . . . 32

4 Experiments and analysis - a standard classification task 33
4.1 Classifier performance 33
4.2 Evolutionary algorithm dynamics 37
4.2.1 An exemplar 37
4.2.2 Sensitivity analysis 38
4.3 Classifier structure 41

5 Experiments and analysis - HTML document classification 45
5.1 Classifier performance 45
5.2 Evolutionary algorithm dynamics 48
5.3 Feature extractor dynamics 48

6 Discussion 51

7 Future work 53

8 Conclusions 56

Bibliography 58

A Additional results 68

B Implementation details and source code 73
B.1 Implementation details 73
B.2 Source code 75
B.2.1 crossvalidate-potter.cpp 75
B.2.2 evolve-pazzani.cpp 79
B.2.3 Classifier.h 83
B.2.4 Classifier.cpp 84
B.2.5 ConceptLearner.h 86
B.2.6 ConceptLearner.cpp 89
B.2.7 DataSet.h 105
B.2.8 DataSet.cpp 107
B.2.9 EvolutionaryAlgorithm.h 115
B.2.10 EvolutionaryAlgorithm.cpp 117
B.2.11 FeatureExtractor.h 123
B.2.12 FeatureExtractor.cpp 125
B.2.13 NaiveBayesianClassifier.h 137
B.2.14 NaiveBayesianClassifier.cpp 138

Chapter 1

Introduction

> "*There is only one thing more painful than learning from experience and that is not learning from experience.*"
>
> Anon.

This dissertation explores the novel application of a biologically-inspired learning algorithm based on the human immune system to the problem of document classification. Its overall aim is to produce a novel, working system built on an immune-based learning algorithm and able to perform better than the currently available learning algorithms. In order to give substance to any claims made, I intend to compare the performance of my system to that of other methods in a systematic and rigorous manner. The motivation for this project is drawn from the current need for techniques which address a range of web-based information retrieval tasks and in this, the introductory chapter, I give a broad overview of the concepts and themes central to the work presented here, laying the foundations on which the work detailed in the following chapters rests.

Concept learning can be framed as the problem of *acquiring the definition of a general category given a sample of positive and negative training examples of the category* [63]. For example, consider the general category of '*papers relevant to this dissertation*', which forms the target concept for which we wish to acquire a definition. Our sets of positive and negative training examples could be the papers listed in the bibliography and those in my rubbish bin respectively. Now imagine a black box, pictured schematically in **Figure 1.1**, with three slots marked '*relevant documents*', '*irrelevant documents*' and '*unknown documents*', a button marked '*learn*', and two round lights labelled '*relevant*' and '*irrelevant*'. Initially, we begin by posting all the papers listed in the bibliography into the '*relevant documents*' slot, and all those retrieved from the rubbish bin into the '*irrelevant documents*' slot.

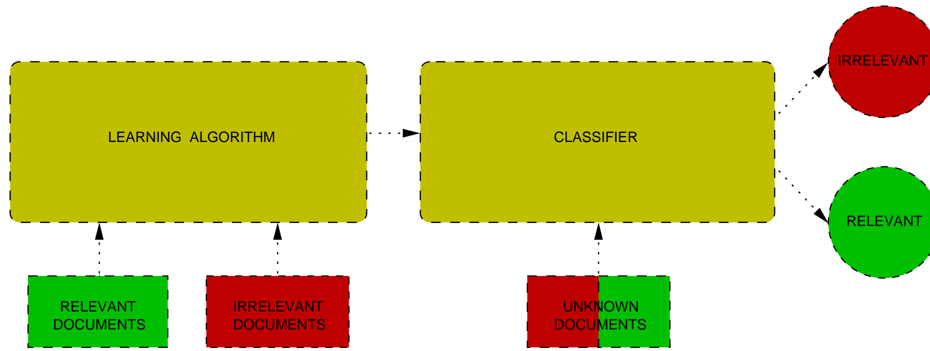


Figure 1.1: An idealised concept learner

After posting all these papers into their appropriate slots we hit the learn button, at which point the machine whirrs and clunks for a few minutes, until finally falling silent. We now turn to a third set of papers whose titles suggest that they may possibly be relevant to the dissertation, and this time feed them into the slot labelled '*unknown documents*'. As we feed each paper into the machine either the '*relevant*' or '*irrelevant*' light comes on, telling us if the paper is indeed relevant or not to the dissertation.

The inner workings of such a black box, at least metaphorically, form the central focus of this dissertation. The black box, from a machine learning perspective, can be seen as a **concept learner**, at the heart of which is a **learning algorithm**, whose job it is to take the training examples and create a **classifier** which is then able to look at further examples and decide if they fit into the learned concept or not. The learning algorithm we implement and study is based on aspects of the dynamics of the human immune system (HIS), part of whose function in its role as protector of the body can be broadly seen as the classification of proteins in the body into two classes: *self* - belonging to the body; and *nonself* - not belonging to the body and potentially harmful. It achieves this classification in part by using a large number of cells called T-cells capable of recognising proteins and produced by its own '*learning algorithm*' in process called **negative selection**, shown in **Figure 1.2**, which results in the survival of only those T-cells which do *not* recognise any cells belonging to the body. The mature T-cells then form the immune systems '*classifier*' and roam the body examining proteins they encounter and destroying those that they *do* recognise. Systems inspired by this and other aspects of the dynamics of the HIS are termed **artificial immune systems** (AISs), and it is one such biologically-inspired system that forms the basis of the learning algorithm and classifier explored in this

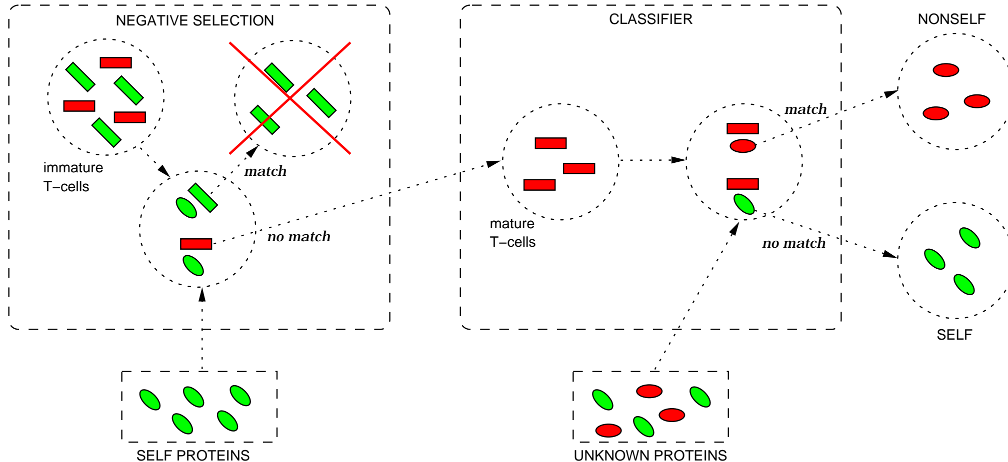


Figure 1.2: Concept learning in the human immune system

dissertation.

The actual generation of the set of T-cells, abstracted as detectors in AIS models, is of course a vastly more complex process than the one just described, and which the AIS implemented here performs through a **cooperative coevolutionary algorithm**, based on previous work by Potter and De Jong [78]. **Coevolution** is *the simultaneous evolution of two or more genetically distinct populations with coupled fitness landscapes* [83], and is often couched within a *competitive* framework in which individual species try to outdo each other in what can be viewed as a type of '*arms race*'. An alternative approach, the one employed here and shown in **Figure 1.3**, is to base the performance of a species on how well it *cooperates* with other species. We evolve several species, each containing a number of detectors, and combine detectors from different species to form our detector set. The success of an individual detector is related to how well the set of detectors in which it is contained *work together*, and not to the detector's individual performance, thereby favouring the evolution of detector sets with detectors which cooperate, rather than compete with each other to achieve some task.

Initially, we test the AIS concept learner outlined above on a standard machine learning data set [85], comparing it to a similar system described by Potter and De Jong [78]. The performance of the AIS is then further assessed on a web-based information retrieval task. Taking the pragmatic definition given by Lancaster [58], we define an **information retrieval system** as one that *does not inform (i.e. change the knowledge of) the user on the subject of his inquiry. It merely informs on the existence (or non-existence) and*

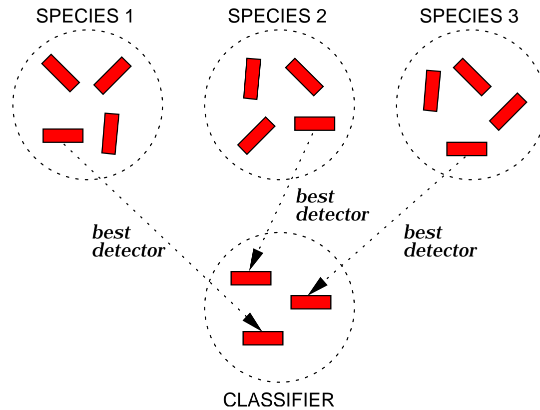


Figure 1.3: Classifier generation by cooperative coevolution

whereabouts of documents relating to his request. Pazzani et al. [72] describe one such system, instantiated as a software agent, that learns a profile of user's interests, which can be equated with a classifier, from a collection of user-rated web pages, and uses this profile to identify other web pages that may be relevant to the user. The agent presents users searching for information with a list of links, called an index page, some of which may be relevant to the user's current interests, some not. Several of these links are visited by the user and rated as relevant or irrelevant, and the agent is then instructed to learn the concepts of relevant and irrelevant on the basis of these user-rated web pages. After learning these concepts, the agent uses them to classify the links on the index page which the user has *not* visited, thereby aiding the user in their search. To construct the concept, Pazzani et al. compare several different standard learning algorithms and find that a naive Bayesian classifier generally performs best. We reimplement this naive Bayesian classifier and in turn compare and contrast its performance with that of the AIS concept learner implemented here.

In applying learning algorithms to the classification of text and HTML documents, the motivation behind this dissertation, an additional concern arises which, although not a central topic of this dissertation, needs to nevertheless be addressed. Learning algorithms usually require data to be structured in the form of **feature vectors** which are composed of a number of features, each with an associated value. For example, a data set describing the physical appearance of a number of people might have a record for each person, with each record containing a number of fields such as height, weight, and so on. These fields represent the features, the values of which together form a feature vector, analogous to the record for a particular person. No

such feature vectors are immediately obvious for a collection of documents, and so a **feature extraction algorithm** needs to first be applied to the documents in order to transform them into feature vectors before they are passed on to the concept learner. One such feature extraction algorithm is implemented in this dissertation.

As already mentioned, the role of this chapter has been to give a broad overview of the concepts and themes central to this dissertation, and to lay out its aims and motivation. **Chapter 2** discusses in more detail key concepts and related work on which the models and methods employed here and detailed in **Chapter 3** are built. **Chapters 4** and **5** describe the experiments performed and the results obtained on the standard and document classification problems respectively, comparing and contrasting the performance of the AIS classifier with that of other learning systems. **Chapter 6** summarises and discusses the implications of the results presented in the previous two chapters, with possible directions for future investigation considered in **Chapter 7**. **Chapter 8** draws the dissertation to a close. **Appendix A** is used to present a number of additional results which, due to their length, would not fit well into the main body of the work. Implementation details and the source code for the programs used to obtain the results of **Chapters 4** and **5** are given in **Appendix B**.

On a final note, a word of caution is necessary: while taking its inspiration from the human immune system, the concept learner presented here makes no attempt or claim to model faithfully or even realistically actual biological actors or mechanisms. Rather, what is sought is a system that exhibits similar functional properties through mechanisms loosely based on current biological understanding. In consequence, any use of biological terminology should be seen as an explanatory aid and not as an indication of a direct correspondence to its biological counterparts.

Chapter 2

Background

> *"I not only use all the brains that I have, but all that I can borrow."*
>
> Woodrow Wilson (1856-1924).

In this chapter fundamental concepts and work related to this dissertation are presented and reviewed, beginning with a look at information retrieval from a general standpoint, and then more specifically at two of its main research areas: feature extraction and concept learning. Practical applications within this field and relevent to this dissertation are then reviewed. A summary of the human immune system is then presented, followed by a discussion of artificial immune systems and their applications. Finally, apposite aspects of evolutionary algorithms are surveyed. The goals of this chapter are twofold: firstly, it represents an attempt to synthesise a large body of work spanning several fields, much of which itself crosses traditional disciplinary boundaries. Secondly, it stands as a platform from which informed decisions can be made concerning the implementation details of the immune-based system of this dissertation.

2.1 Information retrieval

This dissertation focuses on the design and application of an immune-based system for document classification, a task which itself belongs in a wider sense to the field of **information retrieval**. While an exact definition of this field is problematic, from a pragmatic perspective we will adopt, as already mentioned, the one given by Lancaster in terms of information retrieval systems [58]: *an information retrieval system does not inform (i.e. change the knowledge of) the user on the subject of his inquiry. It merely informs*

on the existence (or non-existence) and whereabouts of documents relating to his request. Work in this field has grown steadily since the 1940's and the advent of computers, and has been driven by the need for systems which are able to quickly and accurately access the increasingly large amounts of data being produced and stored on computers. With the birth of the Internet and World Wide Web this need has become more pressing than ever, but the problems of effective retrieval still remain largely unsolved [97].

Much of the work within the field of information retrieval belongs to three main areas: **content analysis**, **information structure**, and **evaluation**. Content analysis is concerned with transforming documents into a form suitable for processing; information structure with improving the effectiveness and efficiency of information retrieval systems through the exploitation of relationships between documents; and evaluation with the assessment of the performance of information retrieval systems. In terms of the concept learner described in the previous chapter, these areas can be equated to deciding what to feed into the machine, how the machine works, and how to assess how well it works respectively. Work relevant to this dissertation concerning the first two of these areas is reviewed and discussed in the following two sections, while the evaluation of system performance is dealt with in **Chapters 4** and **5**.

2.1.1 Feature extraction

In terms of document classification tasks, feature extraction is important in two respects: it can improve classifier accuracy and reduce the amount of computation and storage space needed for classifier training. Much of the work in this area was and still remains centred around the field of natural language processing, whose goal is to analyse, understand and generate languages that humans use naturally, and can be approached from a statistical perspective. This is also the approach taken in this dissertation, and will be described in detail in **Chapter 3**. In recent years however, especially with the birth of the World Wide Web and Internet, several novel approaches have been explored, and it is these approaches that we focus on in the remainder of this section. While these newer approaches are arguably more powerful than the statistical one implemented here, there are several reasons why they have not been used. Firstly, purely from an implementational point of view, they are often much more complex, and, as concept learning as opposed to feature extraction is the main focus of this dissertation, the time spent on implementation would have detracted from that spent on the dissertation's central theme. These newer approaches also produce feature representations which would have been more difficult to integrate into our artificial immune

system model. Lastly, the system with which we compare our concept learner in **Chapter 5** uses a statistical feature extractor, and so in order to provide a more principled comparison we have used the same feature extractor as this system.

Cohen [13], conjecturing that extracting data for learning systems might be qualitatively different from extracting data for other purposes, investigates the role of feature extraction from HTML documents in classifier error. He proposes a feature extraction algorithm based on a semi-automatic wrapper[1] generation procedure [12], and finds that this system decreases classification errors on several learning algorithms. Yang et al. [100] approach the problem of HTML classification from a slightly different perspective by attempting to identify hypertext regularities, such as similar patterns of hypertext links, which exist over a range of domains and which can be exploited in the optimal design of a classifier. They find that the identification and exploitation of such regularities by feature extraction algorithms is crucial for optimal classifier performance as it helps to reduce the amount of noise present in HTML document representations. Ciravegna [10] explores the performance of an adaptive information extraction algorithm for web-based text which is able to automatically adapt to new problem domains. This is useful as the application of information extraction algorithms to new domains usually involves a large amount of specialist information-extraction knowledge at the implementation stage. His system was able to produce significant gains in terms of classification accuracy and reduction of training time of concept learners into which the extracted information was fed, and which form the subject of discussion of the next section.

2.1.2 Concept learning

Concept learning can, in a general sense, be considered as a supervised learning task in which the goal is to infer from a training set a classifier which is able to correctly assign a class to novel examples. Formally, we can define a training set $\mathcal{S} = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_m, y_m)\}$, where each training example $(\mathbf{x}_i, y_i)$ is composed of a vector $\mathbf{x}_i = [x_{i1}, \cdots, x_{in}]$, usually called a feature vector, itself usually composed of discrete or real-valued features x_{ij}; and a class $y_i = f(\mathbf{x}_i)$, drawn from a discrete set of classes $\mathcal{C} = \{1, \cdots, K\}$. A concept learner is a learning system or algorithm which takes a training set as input, and outputs a classifier or hypothesis $h(\mathbf{x})$ which represents an approximation to the unknown target function or concept f. Then, given novel

[1] an interface for a data source that accepts user queries, transforms them into a form appropriate for interrogating the data source, and returns the results to the user.

feature vectors $\mathbf{x}$, the classifier predicts their associated classes $y = h(\mathbf{x})$. In this dissertation, we concentrate principally on a subclass of the general classification problem in which the feature vectors are Boolean, that is $x_{ij} \in \{\text{FALSE}, \text{TRUE}\}$, or, using a binary representation, $x_{ij} \in \{0, 1\}$, and where each feature vector can belong to one of two classes, i.e. $\mathcal{C} = \{0, 1\}$, which can often be interpreted as no/yes, false/true, negative/positive or irrelevant/relevant. In this case, the problem of concept learning can be summarised as one of *inferring a Boolean-valued function from a set of training examples*.

Central to concept learning is the inductive learning hypothesis. While the goal of a concept learner is ideally to determine the hypothesis h which is identical to the target concept f over the *entire* set of possible feature vectors $\mathcal{X}$, also called feature space, the only information available about f is its value over the feature vectors in the training set, which form a subset of $\mathcal{X}$. As no information is available about the remaining unseen members of $\mathcal{X}$, we make the assumption, framed as the inductive learning hypothesis, that *any hypothesis h which approximates the target concept f well over a sufficiently large training set will also approximate f well over the unseen members of $\mathcal{X}$*. The problem of determining the hypothesis h is often viewed as a search problem in which the search space, $\mathcal{H}$, also known as the hypothesis space, consists of possible hypotheses h and the goal of the search is to find the hypothesis which most closely fits the training examples. From this perspective, concept learners can be viewed as search algorithms which are able to effectively search hypothesis space.

Many concept learning systems exist, most employing some form of inductive learning algorithm which arrives at hypotheses by considering specific examples. These algorithms are often based on a symbolic representation language such as predicate calculus [61], decision trees [80] or propositional logic [74], the most popular of such systems including ID3 [79], C4.5 [81], CN2 [11], AQ [61] and ICL [5]. Other approaches such as multilayer neural networks trained with back-propagation [84] and genetic programming [29, 91] have also been used. The performance of many of these algorithms is compared in a number of survey papers [60, 92], and many have found applications in web-based information retrieval task, which we now review.

2.1.3 Applications

In a series of papers, Pazzani et al. [71, 72] explore algorithms for learning and revising user profiles which are then used to determine HTML documents that would be interesting to a user. They compare several learning

algorithms, including a naive Bayesian classifier [63], nearest neighbour [25], decision trees [80] and multi-layer neural networks trained with backpropagation [84], and find that the naive Bayesian classifier generally performed best. They also investigate the role of feature selection in the predictive accuracy of the classifiers, and find that appropriate feature extraction algorithms significantly reduced classification error. They go on to implement the naive Bayesian classifier in a system, *Syskill and Webert*, which automatically filters search results for users. A similar system, *NewsDude* [3], also developed by Pazzani and Billsus, combines a naive Bayesian classifier with a nearest neighbour classification algorithm and is used to recommend news articles. Two user profiles are used in this system, one representing the long-term interests of a user and the other the user's short-term interests created from recently read articles. In this way the recommendation of many similar articles can be avoided. In this dissertation we reimplement the system of Pazzani et al. [72] and compare its performance to our immune-based concept learner, the results of which are presented in **Chapter 5** below.

Several other systems which aid users in the management of web-based information have also been developed. Pant and Menczer [69] describe a webtool called *MySpiders* which consists of an evolutionary multi-agent system that browses adaptively on behalf of users. This tool is designed to complement search engines by creating a large number of agents which autonomously search the internet for information specified by users and which reproduce and die depending on the relevance to the user's query of the information they encounter. A functionally similar system, *WaWa* [26], which allows the construction of web agents for information retrieval and extraction tasks, is based on two neural networks which are trained by user-rated documents. However, *WaWa* is also able to create its own examples through reinforcement learning. The application of reinforcement learning to web-document filtering is also explored by Zhang and Seo [101], who document a system which is initially trained by explicit user feedback and is then refined by observation of user behaviour. They find that their system performs better in terms of information quality and adaptation speed than several other online filtering approaches. Adaptation speed in relation to our system is discussed in **Chapter 7**.

All these systems depend on the existence of both positive and negative evidence, either obtained by explicit user rating of a set of documents, or through the observation of user behaviour and the use of heuristics to infer positive and negative evidence from this behaviour. Schwab et al. [86] propose and test a recommendation system which learns only from positive evidence, thus sidestepping the need for manual rating or heuristics. They found that, as highlighted in the previous section, feature selection played

an important role in increasing the performance of their system, and that the performance of their system was comparable with that of systems which learnt from both positive and negative evidence. Learning from positive examples only has also been explored in artificial immune systems by Hunt and Cooke [49] and in this context will be discussed shortly.

2.2 The human immune system

We now move on to give a brief overview of the major mechanisms and dynamical properties of the human immune system (HIS), of which more detailed accounts can be found in immunology textbooks such as [2, 39] and, from a more artificial immune system orientated perspective, [42]. The HIS plays a key role in maintaining the stable functioning of our body, detecting and eliminating dysfunctional endogenous cells, termed *infectious self*, and damaging exogenous microorganisms, *infectious nonself*, such as bacteria and viruses which enter the body through various routes including the respiratory and digestive systems, and damaged dermal tissues. In a quite different sense, the human immune system also plays a key role in this dissertation, providing inspiration for the fundamental mechanisms on which the immune-based concept learner described in **Chapter 3** is built. This section is perhaps more detailed than might be expected, a fact which we justify by our belief that in order to do effective biologically-inspired computing a detailed understanding of the biological mechanisms from which inspiration is obtained is essential.

The immune system as a whole consists of a multilayered architecture presenting several different lines of defence against infectious material, also termed *pathogens*. The *physical layer* acts to physically block the ingestion of pathogens and includes the skin, nasal hairs, and reflex actions such as coughing and sneezing. The *physiological layer* includes fluids secreted by the body, such as saliva, sweat and tears, which transport pathogens out of the body and contain enzymes that break down pathogenic material. The third layer, the *cellular layer*, is composed of a variety of different cell types with different roles and introduces a further distinction between immune system mechanisms, that of the *innate* and *specific*, *acquired* or *adaptive* immune systems. Innate immunity consists of the defence mechanisms the body utilises immediately or within several hours of infection, and whose response is *non-specific*, that is, the range of pathogens which are capable of generating a response from the innate immune system are fixed at birth for the lifetime of the body, and its level and form of response do not adapt in relation to specific pathogen levels in the body. Adaptive immunity, in

contrast, usually takes several days to become effective and is *specific* in the sense that it adapts to remove a specific pathogenic infection, allocating its resources in a dynamic way.

Most of the cells active in the innate and acquired immune systems belong to the *leukocyte* family, commonly know as white blood cells, and are divided into three major classes: *granulocytes*, *monocytes* and *lymphocytes*, all of which originate from stem cells in the bone marrow. These different cell types are pictured in **Figure 2.1**. Granulocytes, which make up 50% to 60% of all leukocytes, carry granules within their soma containing various chemicals and are themselves divided into three subclasses: *neutrophils*, *eosinophils* and *basophils*. The second class of leukocytes, monocytes, mature into *macrophages*, which play key roles in both innate and adaptive immune system responses. As actors in the innate immune system, macrophages play a similar role to granulocytes in locating and destroying pathogens, and these two cell types are often collectively referred to as *phagocytes*. In the adaptive immune systems, macrophages form members of a functionally grouped set of cells called *antigen-presenting cells* (APCs). While still the subject of ongoing research, it is thought that APCs express *pattern-recognition receptors* (PRRs) on their surface which only respond to material unique to microorganisms not associated with human cells. Upon recognition, the pathogen is ingested by the APC, broken down into protein fragments and reexpressed on the surface of the APC as part of Class II major histocompatability complex (MHC-II). MHC complexes are groups of molecules responsible for the transport of proteins within cells, of which there are two types, the second of which, MHC-I, we shall meet shortly.

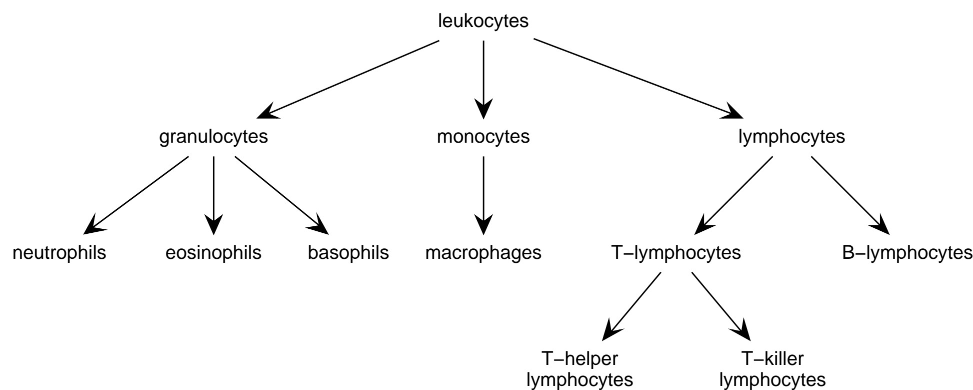


Figure 2.1: Hierarchy of immune system cell types

Lymphocytes, the third class of leukocytes, develop into two major sub-

types - *B-lymphocytes*, or *B-cells*, so called since they mature in an organ called the bursa of Fabricius in birds, the equivalent of which is still unknown in humans; and *T-lymphocytes* (*T-cells*), which mature in the thymus. B-cells mediate *humoral immunity*, the production of *immunoglobulin*, commonly know as *antibody*, a protein which reacts specifically with and neutralises pathogens, and is present on the surface of B-cells. During the recognition stage, these surface antibodies act as receptors which bind with segments of pathogen proteins called *epitopes*, with the strength of the bond between receptor and epitope termed *affinity*. The receptors on the surface of a particular B-cell are all identical, known as *monoclonality*, and if enough of these receptors are able to bind with epitopes above a certain *threshold* level, the B-cell is stimulated by a signal known as *Signal One* or *stimulation* to ingest the pathogen, which it then breaks down into peptides which are reexpressed on the surface of the B-cell as part of a class II major histocompatability complex (MHC-II). The B-cell then takes no further action until it receives *Signal Two* or *help*, from a member of the second type of lymphocyte: T-cells.

T-cells themselves mature into two distinct subpopulations: *T-helper* and *T-killer cells*, the former of which provides the help signal. Maturation of T-cells occurs in the thymus and is a two-stage process. Firstly, immature T-cells, which also possess surface receptors, undergo *positive selection* in which any T-cells with receptors unable to bind with MHC molecules are destroyed. T-cells are then subjected to a process of *negative selection* in which they are exposed to a wide range of *self* proteins and destroyed if they recognise any such proteins. This results in a population of T-cells whose surface receptors only respond to protein fragments, expressed within MHC structures, that are *not* present in self molecules.

Returning to the process of humoral immunity, once B-cells have received *Signal One* and reexpressed peptides of the ingested pathogen on their surface, they need *Signal Two*, provided by T-helper cells to proceed. T-helper cells only produce this signal when two conditions are fulfilled: their surface receptors successfully bind with *nonself* peptides within MHC-II molecules expressed on the surface on B-cells; and they receive a third signal, *costimulation*, which is produced by the binding of an APC to the same T-helper cell of that which is bound to the B-cell. In order to bind to the T-helper cell, the APC must display the same peptides on its surface as those displayed by the B-cell. Once these conditions are met, the T-helper cell will issue *Signal Two* to the B-cell which causes the B-cell to become active. Once active, the B-cell migrates to a lymph node where it divides, producing a large number of B-cells with similar, but not identical, surface receptors in a process called *somatic hypermutation*, the non-hereditable mutation of cell components at

a much higher rate than genetic mutation. Only those B-cells that bind to pathogenic material also transported to the lymph node survive, with their survival rate proportional to their affinity to the pathogen epitopes. Once this process is completed, the remaining B-cells leave the lymph node and differentiate into *plasma* or *memory cells*. Plasma cells produce a soluble form of their surface receptors called *antibody*, which coat pathogens to aid in their destruction by macrophages. Memory cells are B-cells with a long lifespan, possibly up to that of the lifespan of the body, which, through their receptors, are able to recognise and respond to reinfection by the same or similar pathogen much more rapidly in what is called a *secondary immune response*. The entire process of B-cell mitosis, mutation and differentiation is known as *affinity maturation* and *clonal selection*.

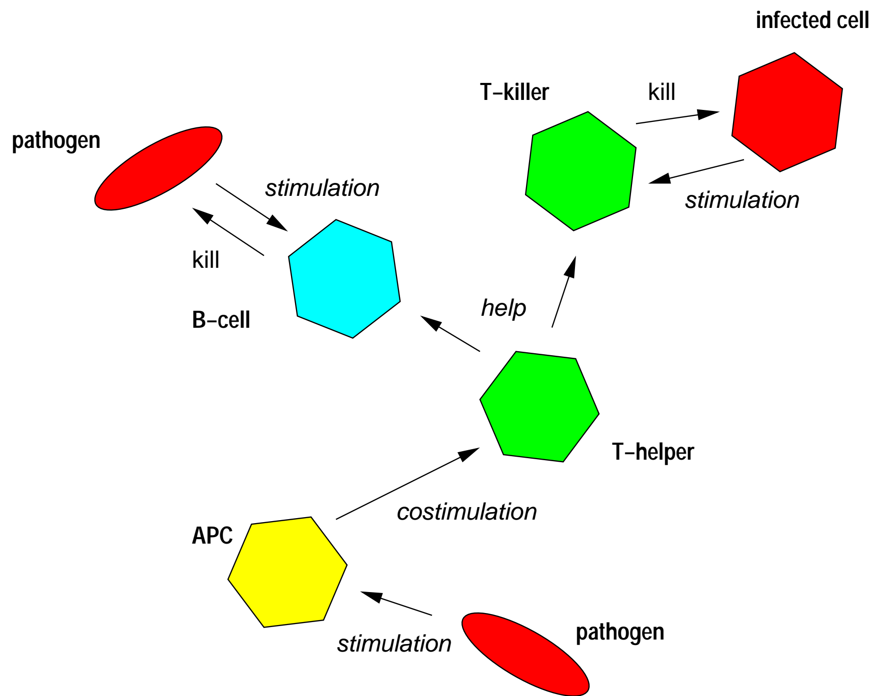


Figure 2.2: The processes of cell-mediated and humoral immunity

A second type of immune mechanism, *cell-mediated immunity*, is mediated by members of the second class of T-cells: T-killer cells. T-killer cells are responsible for the detection and disposal of *intracellular* pathogens, usually viruses, as opposed to extracellular pathogens such as bacteria, which as we have seen are dealt with by B-cells. T-killer cells cannot '*see*' inside cells, but rather rely on a mechanism present in all cells which transports

fragments of proteins inside a cell to its surface and expresses them as part of a MHC-I complex. So, when a pathogen enters a cell, some of its peptides are displayed on the cellular surface. T-killer cells roam the body examining such surface MHC-I complexes, and, if, in a process similar to humoral immunity, they receive a *help signal* from a T-helper cell and match the MHC-I complex, they destroy the cell. The processes of cell-mediated and humoral immunity are summarised schematically in **Figure 2.2**.

2.3 Artificial immune systems

2.3.1 General principles

As seen in the previous section, the HIS employs various techniques in its role of protector of the body. From an information processing perspective, these techniques include *learning*, implemented as affinity maturation and negative selection; *memory*, through crossreactivity and the secondary immune response; and massively parallel and distributed computations using to the order of 10^8 different receptors [96] and 10^7 new lymphocytes produced daily [68]. These all give rise to the general properties of the immune system, which Forrest et al. [31] identify as being: *diverse*, *distributed*, *error tolerant*, *dynamic*, *self-protecting* and *adaptable*. *Diversity* refers to the uniqueness of the HIS both at a population and individual level which, for example, means that different people are susceptible to different pathogens. The fact that there is no central coordination of an immune response, with individual lymphocytes instead controlling such a response, gives the HIS its *distributed* nature. *Error tolerance* relates to the fact that the HIS in general makes very few mistakes, and those which is does make are seldom fatal. The HIS is spatially and temporally *dynamic* as its individual components are constantly being created, destroyed and circulated throughout the body, and *self-protecting* in the sense that the same mechanisms which protect the body also protect the immune system itself. Lastly, it is *adaptable* in that it is able to identify and respond to novel pathogens and also retain a memory of past infections. Artificial immune systems attempt, through the modelling of immune system mechanisms, to exploit one of more of these properties, or, as Dasgupta puts it [16], artificial immune systems *are intelligent methodologies inspired by the immune system toward real-world problem solving.* These definitions draw a distinct line between the fields of artificial immune systems and theoretical immunology, whose goal is to stimulate and complement experimental analysis of the immune system. While the latter is, of course, extremely important, it is the former that is of particular interest to this

dissertation, and so forms the focus of the remainder of this section.

Artificial immune systems can be broadly divided into three categories based on the mechanism they implement: *network-based models*, *negative selection models*, and *lymphocyte models*, although this distinction is a little artificial as many hybrid models also exist. The first of these categories refers to systems which are largely based on Jerne's idiotypic network theory [53, 54], which recognises that interactions occur between antibodies and antibodies as well as between antibodies and antigens, a paradigm explored in more detail in the work of Varela and Coutinho [15, 98]. Negative selection models use negative selection as the method of generating a population of lymphocytes. Lastly, lymphocyte models implement various aspects of T- and B-cell dynamics such as clonal selection and hypermutation in order to create systems with some of the properties described in the previous paragraph. Examples of all of these three types of immune system models can be found in the next section.

Several survey papers exist of artificial immune system methods and applications which offer a much more comprehensive review of the issues discussed in this section. An overview of different immune system paradigms and applications is given by Dasgupta and Attoh-Okine [17], and an up-to-date and comprehensive bibliography of artificial immune system related work is provided by Dasgupta et al. [19]. De Castro et al. [21] present a detailed survey of applications that use immune system metaphors, often combined with other problem solving techniques such as neural networks and evolutionary algorithms. They find applications across a wide range of areas including robotics, optimisation, computer security and machine learning, of which the major areas are briefly reviewed in the following sections. Although diverse, these areas all bear relevance to this dissertation through the insights they provide into the design and application of artificial immune systems.

2.3.2 Applications

2.3.2.1 Machine learning

A number of studies on the general pattern recognition capabilities of immune-based models have been conducted. Forrest et al. [33] describe an immune system model which they use to study the learning and pattern recognition processes that take place in the HIS. The model employs a schematic representation for detectors similar to the one used in the immune-based system of this dissertation, described in detail in **Section 3.1.1**. Their system was assessed on its ability to detect common patterns in noisy input data and to

maintain diversity within its population, both of which are important in the recognition of a large number of patterns with relatively few resources. They found that their system tended to evolve general detectors which were able to recognise a wide range of antibodies through the identification of common schema, an advantageous property in reducing the numbers of detectors necessary for correct classification, and one exploited in this dissertation. Similar studies have also been carried out by De Castro and Timmis [20], and Hunt and Cooke [49], whose system learns from positive examples only. While they demonstrate that learning from positive examples is indeed possible within an artificial immune system context, the test data we use to evaluate our system, as with the majority of classification problems, contains both positive and negative examples. In this case, disregarding the negative examples would be tantamount to throwing away valuable information, and so we choose to implement an artificial immune system which learns from both positive and negative examples. Farmer et al. [28] also approach the study of the general pattern recognition properties of immune system from a dynamical systems perspective and, through insights gained from this approach, are able to identify many similarities with the classifier systems of Holland [46].

Immune-based systems have found several applications in the domain of supervised learning. The AIS classifier of this dissertation, described fully in **Chapter 3**, is largely based on the immune-based concept learner of Potter and De Jong [78], and discussion of this model will therefore be postponed until that chapter. Other such immune-based supervised learning algorithms include that of Carter [7], *Immunos-81*, which models B- and T-cells along with several aspects of their interactions to produce phenomena such as primary and secondary immune responses. Carter assesses his system's performance against several widely-used machine learning algorithms on two standard machine learning data sets, and finds that its performs well. He also experiments with two different strategies for the selection of the B-cells which will reproduce, finding that the best selection strategy to use depends on the data set under consideration. Several studies with particular relevance to this dissertation due to their web-based nature have also been carried out. Cayzer and Aickelin [8, 9] describe and study an immune-based recommender system built on models of antibody-antigen interaction, which provides the matching capability of their system, and of antibody-antibody interaction, which allows diversity to be maintained within the antibody population. They employ their system in two collaborative filtering tasks: the recommendation of movies from a large database, and the prediction of the votes that will be cast for a previously unseen movie. They find that, while the performance of their system without antibody-antibody interactions is satisfactory, the addition of such interactions markedly improves its perfor-

mance. Morrison and Aickelin [65] explore the design and performance of a similar system on the task of website recommendation based on metadata, which differs from the approach taken in this dissertation in that we use the actual data contained in the web page to classify web pages.

Various studies of immune-based paradigms for unsupervised learning tasks such as data clustering and exploratory analysis [52] have also been carried out. Hart and Ross [40] present a system which combines features from immune system and sparse distributed memory [56] models and which is evolved using a coevolutionary genetic algorithm similar to that of Potter and De Jong [78]. This system is used to perform clustering in large, dynamic databases. Timmis et al. [93, 94, 95] compare an AIS model to several other unsupervised learning algorithms including cluster analysis and Kohonen networks. Their system employs a population of B-cells and implements a model of somatic hypermutation to simulate B-cell cloning. They find that this model performs well on several test problems in comparison to other unsupervised learning methods, and also provides an effective technique for exploratory data analysis and visualisation. As a final example of applications to unsupervised learning tasks, Atluri et al. [1] implement an immune-based system incorporating models of memory cells in order to classify remote sensing data on soil moisture levels, a problem involving extremely large amounts of data. They however find that its classification performance is poor compared with that of a neural network classifier, attributing this poor performance to the data representation scheme used and the choice of matching algorithm. This example highlights the need for the careful, problem-specific choice of mechanisms, and perhaps more importantly the fact that, while successful at many tasks, immune-based models are by no means, however rosier a picture is painted by this section, a panacea for all problems.

2.3.2.2 Anomaly detection

Although strictly speaking also a classification problem, the widespread application of AIS models to computer security and more broadly to anomaly detection, and hence the relatively large amount of literature available, make it appropriate to devote a separate section to this area, beginning with a review of applications of immune-based systems to various aspects of computer security.

Somayaji et al. [90] discuss the application of artificial immune systems to computer security from a general perspective, identifying a number of potential uses including the protection of static data, active processes, and networks of mutually trusting computers. Several of these applications are

investigated empirically by a number of authors. Hofmeyr and Forrest [44] describe *ARTIS*, a general artificial immune system which incorporates what they consider three key characteristics of the natural immune system: *robustness*, *adaptivity* and *autonomy*. They successfully apply their system to a network intrusion detection problem, and discuss the relationship between immune classification and another type of classifier paradigm termed learning classifier systems [47]. The parallel between learning classifier systems and immune-based classification is also drawn by Farmer et al. [28], who use a learning classifier system to model the immune system. Forrest et al. [32] describe two systems for detecting anomalous system process behaviour and file-changes associated with the action of a computer virus. The first of these systems initially builds a profile of normal behaviour in terms of system calls for a given set of programs and then uses an immune-based system to monitor the behaviour of these programs and detect abnormal behaviour potentially caused by infection with a virus. As well as changing program behaviour, viruses also change the actual byte-code of the infected program, and this factor is exploited in the second of their systems, which uses an immune-based system to monitor a set of files for such changes. The implementation of such change-detection algorithms is also explored in [34, 35, 41], who use negative selection to generate a population of antibodies. Forrest and Hofmeyr [30, 43] also explore the use of artificial immune systems in network security, outlining a system which monitors network packets and detects abnormal network traffic, which can be associated with attempted intrusions.

While computer security has been the largest area of application of immune-based systems for anomaly detection, Hunt et al. [50] have explored its application in the domain of fraud detection. They describe an artificial immune system designed to automatically detect fraudulent loan and mortgage applications which is able to learn and detect patterns of fraud, and also to incorporate explicit domain knowledge. Their system consists of a population of B-cells which are able to undergo clonal selection and hypermutation.

2.3.2.3 Optimisation

While not '*designed*' for classical optimisation, as with evolutionary algorithms, immune-based paradigms have nevertheless been successfully employed in a number of such optimisation problems. Mori and Tsukiyama [64] develop an adaptive scheduling system which, through the implementation of models of B-cell differentiation and proliferation is able to find good solutions to a scheduling problem with variable batch size. A similar system is implemented by Endoh et al. [27], and found to produce good solutions to the n-city Travelling Salesperson Problem of combinatorial optimisation.

The additional incorporation of an explicit representation of immune system memory cells was found to substantially increase the performance of this system. Huang [48] compare the performance of an immune-based algorithm to that of several other methods on a capacitor placement problem in a radial distribution system[2] and find that the immune-based algorithm offers substantial savings in both computational cost and solution quality.

Central to all the systems described above is the need for a mechanism which is able to generate and maintain lymphocyte populations. We now move on to first examine from a general perspective one such possible method, that of evolutionary algorithms, and then to address specifically how this paradigm can be used in the creation of concept learners and how immune-inspired mechanisms can enhance the performance of evolutionary algorithms in general.

2.4 Evolutionary algorithms

The complexity of many computational problems, whose solution often involves searching through a vast number of possible answers, has led to the development of a range of innovative techniques. One area of research which has attracted a large amount of interest in recent years is that of evolutionary computation, within which evolutionary strategies [82], genetic programming [57] and genetic algorithms [45] form the main threads. The central idea behind these three approaches, collectively referred to as evolutionary algorithms, is the evolution of a population of candidate solutions through the application of operators inspired by natural selection and random variation. Evolutionary algorithms have been successfully applied in the solution of many practical problems in a wide range of fields including engineering, machine learning, computer science and economics, as well as offering important insights into the dynamics of natural evolutionary systems. We focus here on those aspects of evolutionary algorithms applicable to this dissertation, namely that of coevolution and the evolution of concept learners, and leave to reader to consult the many excellent texts, such as [38, 62], for a more general introduction.

2.4.1 Cooperative coevolution

In a series of papers Potter and De Jong [75, 76, 78] and Potter, De Jong and Grefenstette [77] explore the use of a cooperative coevolutionary algorithm

[2]an electrical system which distributes power from one generating source to a number of targets and where, if there is a power failure, all targets lose power.

for function optimisation and for the evolution of artificial neural networks, sequential decision rules, and learning algorithms. Their approach is the one taken in this dissertation and described fully in **Chapter 3** below. Briefly, it involves the evolution of a number of non-interbreeding subspecies, individuals of which only represent partial solutions to the problem at hand, and are combined to form a complete solution. Sofge et al. [89] extend this approach in an attempt to decrease the degree of epistasis which can sometimes occur in cooperative coevolutionary approaches. Their system involves the '*blending*' of the usually distinct species of individuals as evolution proceeds, which they find helps the population to escape local optima. Neri [66] also investigates the incorporation of cooperative coevolution into three learning algorithms and shows that such algorithms are able to produce efficient concept descriptions. Concept learners have also been created using a variety of other evolutionary techniques, a survey of which is briefly made in the next section.

2.4.2 Evolving concept learners

In [22], De Jong et al. explore the use of genetic algorithms in the design and implementation of concept learning systems. Their supervised concept learner, *GABIL*, is based around a standard genetic algorithm which searches a space of classification rules for a set which performs well on a given classification problem. Their approach to evolving solutions is however somewhat different to standard techniques in that it uses a *batch-incremental* training regime in which the concept learner is initially evolved to classify just one training example from the training set. Once *GABIL* has learnt to classify this example correctly, another is presented and, if classified correctly, no evolution takes place and a further training example is presented. If the new training example is incorrectly classified, the system is evolved until it correctly classifies this and all previously presented examples correctly. They show that the performance of this incremental approach to rule set evolution compares favourably to that of other standard concept learning algorithms and also offers a means of dynamically adjusting biases inherent in concept descriptions in order to further improve classifier performance.

Other approaches to the evolution of classifiers have also been explored. Folino et al. [29] explore the evolution of decision tree classifiers using a cellular genetic programming approach and consider two different evolutionary approaches: a coarse-grained island model [14], and a fine-grained diffusion model [73]. They opt for the fine grained diffusion model, arguing that this method enables faster convergence and improved accuracy. Genetic programming as a method for the evolution of classification rules is also explored by

Tan et al. [91]. Dasgupta and González [18] evolve complex fuzzy classifier rules using a canonical generational genetic algorithm with a direct encoding of the rules and various special operators. Their results are compared in **Chapter 4** with those obtained for the learning systems explored in this dissertation.

Forrest et al. [33], as already mentioned in **Section 2.3.2.1**, describe an immune system model which they use to study the learning and pattern recognition processes that take place in the immune system and in which a genetic algorithm plays a central role. In order to maintain population diversity in their system they employ an algorithm utilising emergent fitness sharing and similar to the bidding method used by learning classifier systems [47], which, further to their theoretical exploration in [88], they compare empirically with explicit fitness sharing [23]. This maintenance of diversity in the human immune system has also directly inspired a number of extensions to the canonical genetic algorithm, which we now briefly discuss.

2.4.3 Immune-based evolutionary algorithms

The maintenance of genetic diversity within a population is necessary for the long term success of an evolutionary system, as without such diversity populations would become trapped in local optima and unable to respond to environmental changes, termed *premature convergence*. The human immune system, as we have seen in **Section 2.2**, is an example of a system which maintains a population of diverse individuals, and has provided the inspiration for a number of artificial evolutionary systems. In [87], Smith and Forrest analyse a immune-based cooperative genetic algorithm from the perspective of diversity maintenance, also contrasting it to explicit fitness sharing mechanisms [23], and find that, as well as effectively maintaining genetic diversity, their algorithm also produces more general solutions than genetic algorithms employing explicit fitness sharing. Jiao and Wang [55] also describes an immune-based genetic algorithm which extends the canonical genetic algorithm through the introduction of two new genetic operators based around the concepts of vaccination and clonal selection. They test their extended algorithm on a 442-city Travelling Salesperson Problem and find that it obtains the optimal solution in an order of magnitude fewer generations than the canonical algorithm.

This chapter has, through necessity, been rather lengthy. We are now, however, in a position to make informed choices concerning the construction of our own immune-based concept learning system, which forms the subject matter of the next chapter.

Chapter 3

Materials and methods

> *"In preparing the soil for planting, you will need several tools. Dynamite would be a beautiful thing to use, but it would have a tendency to get the dirt into the front-hall and track up the stairs."*
>
> Robert Benchley (1889-1945).

In this chapter we describe in detail the materials and methods used, beginning with the two classifiers implemented - the artificial immune system (AIS) classifier, and the naive Bayesian classifier (NBC). Descriptions of the cooperative evolutionary algorithm and test data then follow, and the chapter ends with an account of the feature extraction algorithm employed.

3.1 The classifiers

In this section we describe the two classifiers which we will presently use in our experiments in **Chapters 4** and **5**. They have commonalities in so far as they both attempt to determine the class of an example by examining relationships between the attributes of the example's feature vector. However, they diverge in the mechanisms by which they do this, one based on immune-system principles, the other on statistical analysis. We begin by describing the immune-based classifier.

3.1.1 The artificial immune system classifier

The AIS classifier is based on one described by Potter and De Jong [78], and is composed of a set of detectors, each of which is instantiated as a ternary schema of the same length as the feature vectors it will classify. Associated

with each detector is a real-valued threshold which indicates the percentage of matching bits between schema and feature vector necessary before a match is said to have occurred. The strength of the match between detector and feature vector is the percentage of matching bits in the schema and feature vector, ignoring any positions where the schema contains a #. For example, for the detector and feature vector shown in **Table 3.1**, there are 2 matching bits out of 5 non-# bits, so the binding strength between the detector and feature vector is $\frac{2}{5} = 0.4$. The calculated binding strength must be greater than the threshold of the detector to consider a match to have occurred. Detectors can be of one of two types, Type 0 or Type 1, with a Type 0 detector, as in the human immune system, classifying any feature vector it matches as nonself, while a Type 1 detector contrarily classifying matching feature vectors as self.

detector	`01#1##11`
feature vector	`11100101`

Table 3.1: Example detector and feature vector

3.1.2 The naive Bayesian classifier

The naive Bayesian classifier [25, 63], a probabilistic method of classification, calculates the probabilities of a particular feature vector belonging to each possible class and then classifies the feature vector as belonging to the class for which this probability is highest. Formally, if $\mathbf{a} = [a_1, a_2, \cdots, a_n]$ is a feature vector made up of n features, a_i, and $V = \{v_1, v_2, \cdots, v_m\}$ is a set of m classes, then the class $v_{NB} \in V$ that the NBC classifies the example $\mathbf{a}$ as belonging to is given by:

$$v_{NB} = \operatorname*{argmax}_{v_j \in V} P(v_j) \prod_{i=1}^{n} P(a_i|v_j), \ \forall \ v_j \in V$$

On a practical level, the NBC was easy to implement as the work of calculating the conditional probabilities had already been done during the feature extraction process which will be described in **Section 3.4** below.

3.2 The evolutionary algorithm

The scheme used to evolve an AIS concept learner is based on a coevolutionary approach described by Potter and De Jong [78] briefly outlined in the

last chapter. The cooperative coevolutionary algorithm consists of a number of non-interbreeding species of detectors, whose encoding will be described shortly, and initially starts with one randomly initialised species whose fitness is evaluated as described below. The initialisation of species is controlled by two parameters: a *generality bias* parameter and a *type bias* parameter, both in the range $[0, 1]$. The generality bias parameter represents the probability that any position in a newly initialised detector contains a #, as opposed to a 0 or 1. The type bias parameter is the probably that a detector will be of Type 1.

At each generation, a trial population composed of the fittest detector in each species is created and the fitness of this trial population evaluated. The fitness of all individuals in a species is then evaluated, as described below. Next, child species are created by selecting two parents from the same species using fitness-proportionate selection with balanced linear scaling [62], which are then recombined using uniform crossover, and mutated by bit flipping to create a child detector, which forms part of the child species for the species the parents were selected from. This process continues until the child and parent species are the same size. The fitness of each individual in the new species is then evaluated. If the fitness of the trial population fails to increase above a certain stagnation threshold over several consecutive generations, a new species is added and any species not contributing to the fitness of the trial population are removed. Values for the parameters used in the experiments described below are given in **Table 3.2**. A schematic representation of the control flow of the evolutionary algorithm is given in **Figure 3.1**.

parameter	**value**
species size	100
crossover rate	0.6
mutation rate	$\frac{2}{\text{genome length}}$
stagnation threshold	0.001
stagnation generations	2
generality bias	0.5
type bias	0.5

Table 3.2: Evolutionary algorithm parameter settings

3.2.1 The encoding scheme

Detectors are encoded as binary genomes each containing 4 genes, as pictured in **Figure 3.2**. The first gene, the 8-bit threshold gene, encodes the value

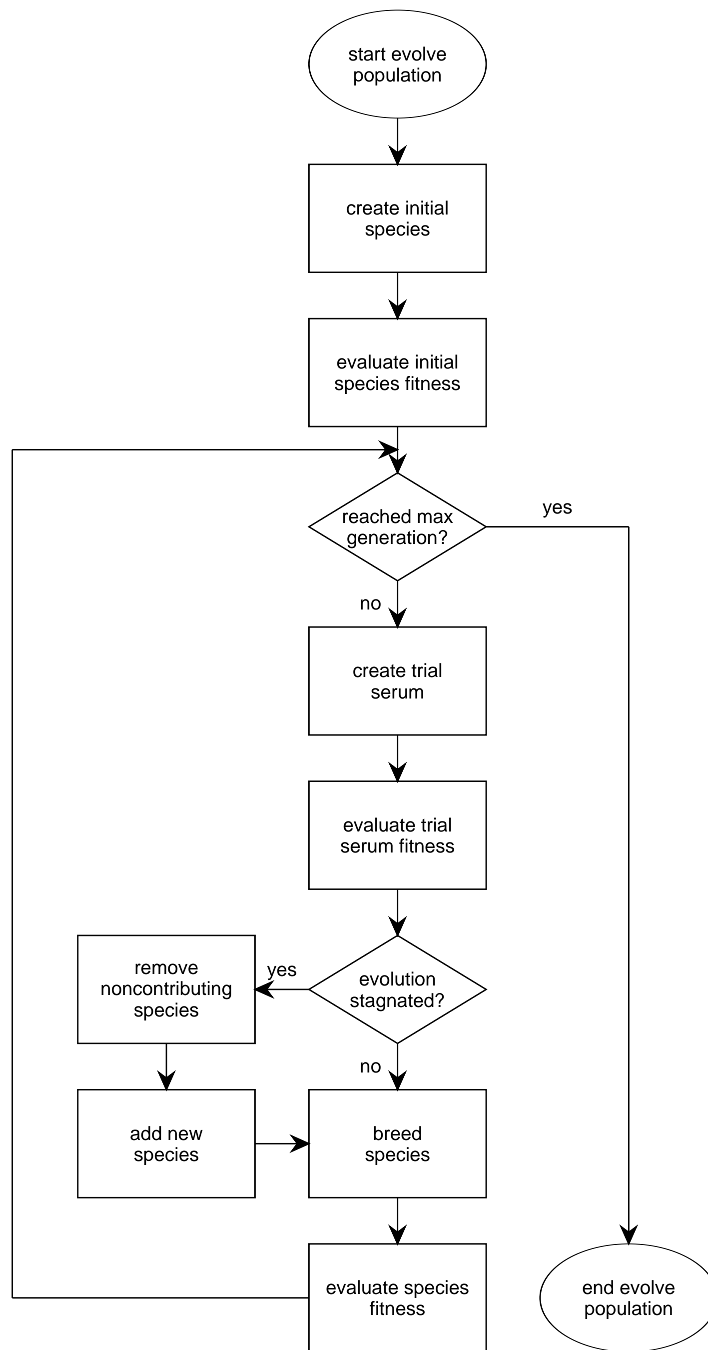
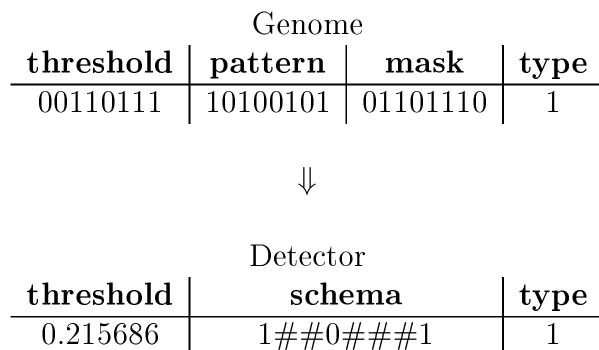


Figure 3.1: Control flow of evolutionary algorithm

for the detector's threshold. A Gray coding was used for this gene in order to avoid small changes in the genotype producing disproportionately large changes in the phenotype. The threshold value is calculated by converting the gene to base 10 and then dividing this value by 255 to get a real number in the range $[0, 1]$. The second and third genes, the pattern and mask genes, are combined to form the detector's schema. Each of these genes has the same number of bits as the number of bits in the feature vectors the AIS classifier is designed to operate on. The mask gene is overlaid onto the pattern gene and any positions at which the mask gene is 1 changes the corresponding bit in the pattern gene to a #. A value of 0 in the mask gene leaves the corresponding bit of the pattern gene unchanged. In this way the schema is formed by copying the pattern gene, modified by the mask gene. The fourth gene stores the detectors type, as previously described in **Section 3.1.1**.



Genome

threshold	**pattern**	**mask**	**type**
00110111	10100101	01101110	1

⇓

Detector

threshold	**schema**	**type**
0.215686	1##0###1	1

Figure 3.2: Detector encoding scheme

This is of course only one of many possible encodings and perhaps even seems a rather inelegant way of shoehorning the problem into a binary representation. We therefore also ran several experiments with a direct encoding which encoded the threshold gene as a real value in the range $[0, 1]$, the schema as a ternary 8-tuple, and the type gene as a 1-bit binary value. The threshold gene was mutated using Gaussian mutation and no crossover operator was applied to this gene, while mutation of the schema gene was achieved by randomly changing it to one of its two other possible values. Mutation of the type gene was handled by standard bit flipping and a uniform crossover operator was applied to the schema and type genes. It turned out that neither encoding scheme produced significantly better performance in our experiments, so we opted to continue using the encoding of Potter and De Jong in order to simplify comparisons between our system and theirs.

3.2.2 The fitness evaluation scheme

The fitness of the trial population, composed of the best individual from each species, is calculated by presenting it with each training vector in the training set in turn. The detector in the trial population which matches the current training vector with the greatest binding strength is then found, and if this strength is greater than the detector's threshold, the detector is said to have matched the training vector, and assigns it to *Class 1* (or *Class 0* if the detector is Type 1), otherwise if no match occurs the training vector is assigned to *Class 0*. The assigned class is then compared with the actual class of the training vector, and if equal the trial population is said to have classified the training vector correctly. The number of correct classifications made by the trial population over the entire training set is recorded and converted into a percentage of the total number of training vectors to give the predictive accuracy of the trial population on the training set. The fitness of all individuals in the trial population is then set to this value, and the individuals are reinserted into their respective species. A schematic representation of the control flow of the fitness evaluation scheme is given in **Figure 3.3**.

3.3 Test data

3.3.1 The voting problem

Two sets of test data were used in the experiments of **Chapters 4** and **5**, both taken from the UCI Repository of Machine Learning Databases [4]. The first data set, the 1984 United States Congressional Voting Records [85], gives the voting records for 267 Republican and 168 Democrat members of the U.S. House of Representatives. Each record holds the vote cast by the member on 16 different issues, and the original records have been simplified to record this vote as yea, nay or abstain. Each member's voting record is converted to a Boolean feature vector, with each consecutive pair of bits representing a vote for a particular issue using the scheme given in **Table 3.3**. Associated with each record is the class the record belonged to: 0 for Democrat, 1 for Republican.

3.3.2 The HTML document classification problem

The second data set, the Syskill and Webert Web Page Ratings [70] consists of 4 data sets: Bands, BioMedical, Goats and Sheep, each containing HTML pages related to a particular topic. A user rated each page in a set as not

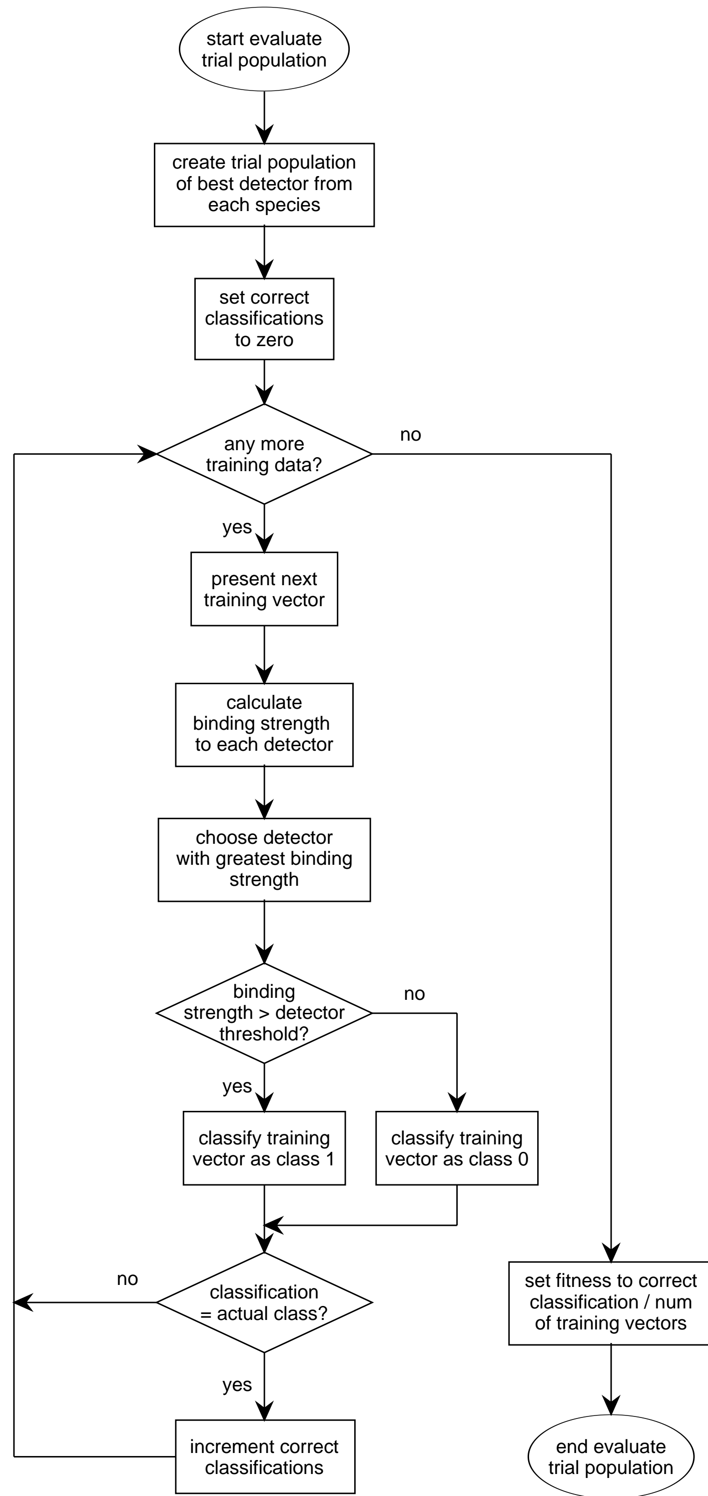


Figure 3.3: Control flow of fitness evaluation scheme

vote	**encoding**
abstain	00
yea	01
nay	10

Table 3.3: Encoding scheme for voting data set

interesting or interesting, which allowed a page to be assigned to one of two classes: Class 0 (*cold*) and Class 1 (*hot*) respectively. The task in this problem is to make predictions about whether examples from an unseen set of web pages would be interesting or not from the information contained within a training set of ranked pages. **Table 3.4** provides a summary of the structure of the data sets.

data set	**total examples**	**number of classes**	**number of attributes**	**number of categories**
Voting	435	2	16	3
Bands	61	2	N/A	2
BioMedical	131	2	N/A	2
Goats	70	2	N/A	2
Sheep	65	2	N/A	2

Table 3.4: Data set summary

3.4 The feature extractor

Unlike the first data set, which offered a fairly straightforward encoding of the raw data into feature vectors, for the document classification problem such a simple encoding is not so obvious when dealing with HTML input data. Following Pazzani et al. [72], a feature extraction algorithm was used to convert a raw HTML document into a Boolean feature vector. Each bit in the feature vector represents the absence or presence (at least once) of some associated feature, in this case a word, in the document. The task of the feature extraction algorithm is to decide from which words to compose the feature vector, and this is done using an information-based approach to extract the most informative words from a collection of documents [59, 72].

Initially, the feature extraction algorithm takes the complete set of pages, S, and creates a list of all the words, W, contained in the pages. If a word,

considered as a sequence of upper of lower case letters (a-zA-Z) separated by nonalphabetic characters, occurs more than once on the same page or across several pages, it is only represented once on this list. All words were converted to upper case and any words occurring on a list of frequently used words (**Table 3.5**) were removed. The *expected information gain*, $E(w, S)$, that the presence or absence word $w \in W$ gives towards the classification of S [80], is:

$$E(w, S) = I(S) - [P(w = \text{pres})I(S_{w=\text{pres}}) + P(w = \text{abs})I(S_{w=\text{abs}})]$$

with $P(w = \text{pres})$ is the probability a word is present at least once on any page, $S_{w=\text{pres}}$ the set of pages containing the word w, and,

$$I(S) = \sum_{C \in \{\text{hot,cold}\}} -P(S_C) \log_2[P(S_C)]$$

where S_C is the set of pages belonging to class C, and $P(S_C)$ is the probability of a page belonging to that class.

In the above equations $I(S)$ can be seen as representing a measure of the amount of information that knowing which class a document was in would give us. If there was a fifty-fifty chance that a document was in either class, $I(S)$ would be 1 and we would have gained 1 bit of information. If all the documents were in one class, $I(S)$ would be 0, and no information would have been gained as we would already know which class the document was in. For a skewed distribution of documents into classes, the value of $I(S)$ lies between these maximum and minimum values. The second term in the equation for $E(w, S)$ above, which is subtracted from $I(S)$, is basically a measure of how much more information we need in order to decide on the class of a document given that a particular word is present or absent in that document. Thus, if a particular word tells us which class a document is in, then this second term is minimised and $E(w, S)$ maximised. On the other hand, if the presence or absence of a word provides no information as to which class a document is in, the second term is maximised and $E(w, S)$ minimised. In summary, the higher the expected information gain, $E(w, S)$, for a particular word, the more information it provides in deciding which class a document belongs to, and the more informative that word is considered. Therefore, to create n features the extraction algorithm uses the n words with the highest values of $E(w, S)$. Each HTML document is then converted to a Boolean feature vector by assigning a 1 to the appropriate feature if the document contains the word at least once, and a 0 if the document does not contain the word.

AND	HREF	THE	IMG	SRC
FOR	FONT	COM	ALIGN	ALT
SIZE	INDEX	HTM	TITLE	GOPHER
ORG	NAME	THIS	WEB	YOU
WWW	HOME	ABOUT	INTERNET	WIDTH
PAGE	FTP	BODY	ARE	LIST
HTML	NET	HEIGHT	LINKS	NEWS
FROM	HEAD	STRONG	WELCOME	WITH
TOP	MAILTO	YOUR	GIFS	BOTTOM
MAIL	CGI	THAT	BIN	ALL
CENTER	WUSTL	GDB	GOV	OTHER
ANY	HAS	NOT	TOC	GNN
HTTP	GIF	WIC	SERVER	AVAILABLE
IBC	ADDRESS	INFORMATION	HERE	CAN
EDU	WHAT	MORE	OUR	WILL
HAVE	COMMENTS	WHO	PLEASE	ALSO

Table 3.5: Frequent words removed from the word list

3.5 Summary

We have now reached somewhat of a halfway point in this dissertation, and so it seems an opportune moment to briefly review what has been achieved so far, and to point the way forward to what remains. We initially gave a broad outline of the context in which the work present here sits, and then went on to review and discuss at some length work related to that of this dissertation. In this section we have given a detailed description of the system we chose to implement in light of this related work, and also of the systems and data which we will use as a benchmark in a number of experiments and against which we will evaluate our own immune-based concept learner. We now go on to describe and present the results of these experiments, and to then discuss these results and consider directions for future work.

Chapter 4

Experiments and analysis - a standard classification task

> *"The materials of action are variable, but the use we make of them should be constant."*
>
> Epictetus (c. 50-138 A.D.)

Our goal in this section is to obtain an understanding of the dynamics of the immune system and naive Bayesian classifiers on the voting classification problem, and to compare and contrast their performance. To do this, we conduct a series of experiments which highlight different aspects of the classifiers' performance. Initially, in **Section 4.1**, we calculate the predictive accuracy of each classifier. In **Section 4.2** we take a more detailed look at the dynamics of the cooperative evolutionary algorithm, examining the effects a range of parameter settings have on classifier performance. Finally, in **Section 4.3**, the structure of the classifiers produced by the two learning algorithms is studied.

4.1 Classifier performance

One of the most commonly used measures of classifier performance is that of **predictive accuracy**. The true predictive accuracy of a classifier on a particular problem, represented as a data set, is the probability that the classifier will classify any randomly chosen example correctly, or, alternatively, is the probability that the classifier will *not* make an error in classifying the example. For example, in order to achieve a perfect score of 1.0 on the voting problem, a classifier would have to, given any voting record, be able to say correctly whether the record was that of a Democrat or Republican.

In an *a priori* world, an exact figure for this metric could be obtained by testing the classifier on all possible voting records. However, as with many real-world problems, since the voting records given are just those from 1984 and therefore only a *sample* of all possible voting records over all years, we have to appeal to statistics for an estimation of the classifier's true predictive accuracy. The **sample predictive accuracy**, a_s, of a classifier is defined as:

$$a_s = \frac{correctly\ classified\ examples}{total\ number\ of\ examples}$$

and, given an appropriate experimental setup, is an unbiased estimator of the true predictive accuracy.

For the voting problem, **10-fold crossvalidation** is the recommended procedure for calculating the sample predictive accuracy as an unbiased estimator of the true predictive accuracy of a classifier [99]. 10-fold crossvalidation involves randomly dividing the complete data set into 10 equally sized disjoint sets, and then using 1 subset as a test set and the other 9 as a training set. The training set is used by the learning algorithm to create a classifier, whose sample predictive accuracy is then calculated using the test set. This process is repeated for each of the 10 subsets, the mean sample predictive accuracy of these 10 trials forming a unbiased estimator to the true predictive accuracy of the classifier. These results can be further refined by repeating the 10-fold crossvalidation process a number of times and averaging the predictive accuracy over these trials.

In our experiments we estimated the predictive accuracy of the immune system and naive Bayesian classifiers by performing 10-fold crossvalidation over 5 trials. A randomly constructed crossvalidation set was used in each trial and the trials were **paired**, meaning the same training and test sets were used to train and test the two classifiers on each iteration. The results of these experiments are shown in **Figure 4.1** and summarised in **Table 4.1**. **Figure 4.1** is a density plot of the distribution of predictive accuracies of the classifiers produced in the crossvalidation trials. Density plots can be thought of as histograms with a large number of bins, producing a smoother representation of the distribution of results over a number of trials. Predictive accuracy is represented along the x-axis, and the relative frequency with which a classifier with this predictive accuracy was observed during the experiments along the y-axis. Since each 10-fold crossvalidation run produces 10 classifiers, and this was repeated 5 times, a total of 50 measures were obtained of the predictive accuracy of classifiers produced by each learning algorithm.

For classifiers produced by the naive Bayesian algorithm, **Figure 4.1** shows a right-skewed unimodal distribution with a single low, spread peak

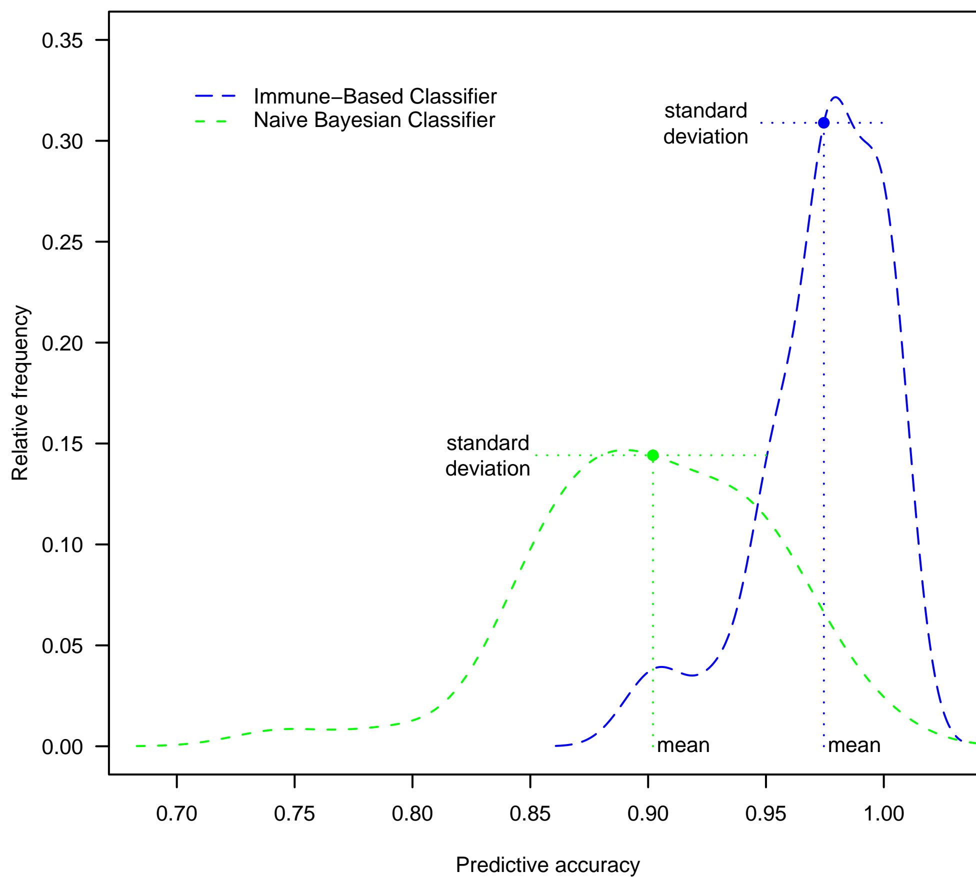


Figure 4.1: 10-fold crossvalidation (voting data set)

Algorithm	**predictive accuracy**	**standard deviation**	**95% confidence interval**
immune-based	0.974	0.026	0.057
naive Bayesian	0.901	0.049	0.088

Table 4.1: Summary: 10-fold crossvalidation (voting data set)

(standard deviation: 0.049) almost symmetrical about its mean predictive accuracy of 0.901. In contrast the immune system concept learner produced a less symmetric distribution with a higher mean of 0.974, rising fairly steeply and then dropping off even more steeply, giving a tighter distribution of values (standard deviation: 0.026) than the NBC. To determine if there was any statistically significant difference between the two distributions we performed a Wilcoxon rank sum test, the results of which indicated that the means of the distributions are indeed different. 95% confidence intervals for these means are also given in **Table 4.1**. From these results we can conclude that the immune-based learning algorithm repeatedly produced classifiers which were able to correctly classify more examples than the classifiers produced by the NBC, and that any given classifier produced by the immune-based algorithm is more likely to be closer to the true predictive accuracy than that of an NBC.

These results can also be compared to others reported in the literature for a number of different classifiers and summarised in **Table 4.2**. These results were calculated on the voting problem using the same 10-fold cross-validation testing regime as the one employed here, with the exception of Lim et al. [60], who, instead of reporting predictive accuracy, reported on the rate of misclassification, e_s, of the algorithms they tested. This statistic is also known as the classifier's sample error rate and defined as the number of misclassifications made by the classifier, i.e. $e_s = 1 - a_s$, where a_s is the classifier's predictive accuracy. Other statistics such as variance and confidence intervals are also given where provided by the original paper, and these statistics are also reproduced for the two learning algorithms of this dissertation for ease of comparison. The predictive accuracy, 0.964, of the AIS concept learner of Potter and De Jong [78], calculated using one 10-fold

Algorithm	**predictive accuracy**	**standard deviation**	**95% confidence interval**	**error rate**
immune-based	0.974	0.026	0.057	0.026
AIS [78]	0.964		0.018	0.036
QUEST [60]	0.963			0.037
AQ15 [78]	0.956		0.023	0.044
POLYCLASS [60]	0.948			0.052
Fuzzy Classifier [18]	0.947	0.316		0.053
naive Bayesian	0.901	0.049	0.088	0.099

Table 4.2: Comparison of classifier performance (voting data set)

crossvalidation run, is lower than that of the similar immune-based classifier implemented in this dissertation. They also reported on the performance of classifiers produced by the AQ15 symbolic inductive learning system and found, with a mean of 0.956 and 95% confidence interval of 0.023, that there was no significant difference in its performance compared to that of their AIS concept learner. Dasgupta and Gonzalez [18] evaluated an evolved fuzzy rule-based classifier on a 10-fold crossvalidation regime over 5 trials and reported its predictive accuracy as 0.947 with a variance of 0.1, while Lim et al. [60] tested 33 classification algorithms, finding the POLYCLASS and QUEST algorithms to have the highest predictive accuracy of 0.948 and 0.963 respectively, although their performance was not statistically significant from that of 20 other algorithms they also tested. While data to perform statistical tests[1] which would give a more quantitive analysis of the differences in these classifiers and the ones trialed in this dissertation was not available, the difference in the predictive accuracy of the immune-based algorithm and that of the classifiers reported in **Table 4.2** suggests that the immune-based algorithm outperforms all of these algorithms.

4.2 Evolutionary algorithm dynamics

4.2.1 An exemplar

The previous section gave an indication of the predictive accuracy of the classifiers produced by the AIS concept learner and compared this to the performance of several other algorithms, but gave little insight into the dynamics of the system which produced these classifiers. **Figure 4.2** consists of several graphs depicting the evolution of a typical classifier over 100 generations of one crossvalidation run of the last section. These graphs show, from top to bottom, the predictive accuracy, classifier size and mean predictive accuracies of the individual species from which the classifier's detectors were drawn. The vertical red line at generation 69 identifies the point at which the classifier with the highest predictive accuracy was found. Initially, the classifier starts with one species, whose mean fitness increases until generation 9, when a new species is created due to the stagnation of the predictive accuracy of the classifier. The classifier's size then stays at 2 until generation 45, although, as seen by the relatively large downwards steps at generations 29 and 36 for species 2, since the classifier's predictive accuracy has stagnated, the individuals of the second species are randomised through the deletion and

[1] a t-test would have produced unreliable results as our data violates one of the assumption of this test, namely that the sample distribution is normally distributed.

then recreation of this species. The size of the classifier increases steadily from 2 to 6 species over generations 36 to 54, where it stays until generation 69, although species 6 is randomised several times during this period. At generation 68 the classifier's size is reduced to 5, and this is in fact the generation at which the best classifier found over the entire run is produced. From then on the size of the classifier oscillates between 5 and 6, with species 5 and 6 periodically being destroyed and recreated, as shown once again by the relatively large vertical steps in the mean fitness of these two species, the periodicity of 2 generations resulting from the setting of the *stagnation generations* parameter to 2. From these graphs and our observations of the evolutionary algorithm over many runs we can conclude that, generally, the number of species steadily increases, retaining good detectors, until a size is reached when further increases in the number of species generally offers no improvements in performance. At this point the algorithm seeks to reduce the number of species by repeatedly destroying and recreating the most recently added species.

4.2.2 Sensitivity analysis

In order to gain a more empirical understanding of how changes in the configuration of the AIS concept learner affected the performance and composition of the classifiers it produced, we repeated the 10-fold crossvalidation experiment described in the last section with a variety of differently configured concept learners. As well as varying the number and size of the initial species used in the coevolutionary algorithm, we were also interested in seeing if the manner in which its constituent individuals were initialised affected classifier predictive accuracy and size. **Figure 4.3** shows a series of density plots which, as in **Figure 4.1**, depict the distribution of classifier predictive accuracy over five 10-fold crossvalidation trials in which different values for the generality and type bias parameters, described in **Section 3.1.1**, were used. These nine plots have been placed on a grid with generality bias increasing horizontally and type bias vertically. The top left corner of this grid depicts the distribution for a classifier where both parameters were set to 0.1, and the bottom right a classifier with both parameters set to 0.9. The central rows and columns contain graphs for classifiers with a type and generality bias of 0.5 respectively. As can be seen from these graphs, generality biases of 0.1, that is, allowing very few # values in newly initialised individuals, on the whole marginally decreased the predictive accuracy of evolved classifiers. Type biases, the probability that a detector will be of Type 1, seemed to have little effect on overall classifier performance, and we concluded that, in general, the evolutionary algorithm seemed to be fairly robust to the settings

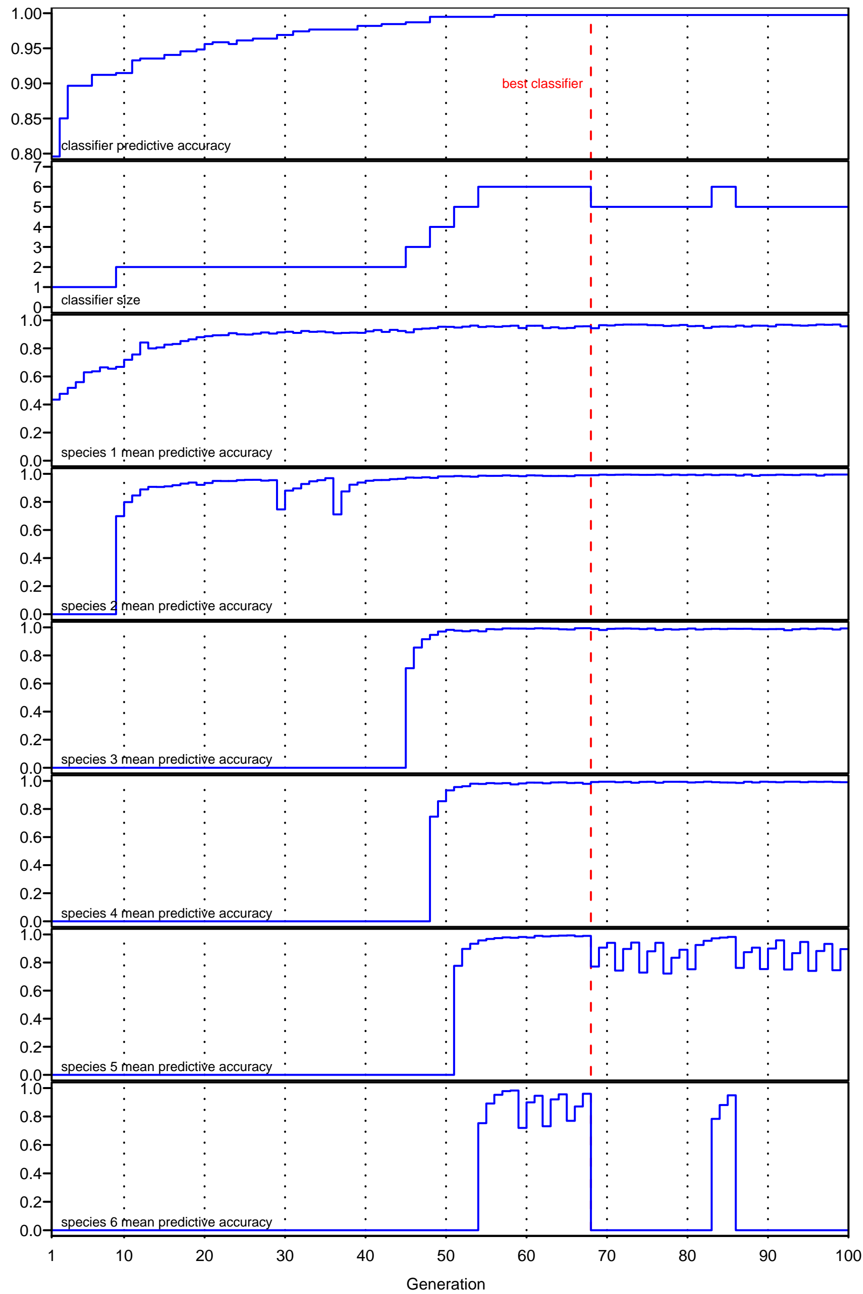


Figure 4.2: Evolutionary algorithm dynamics (voting data set)

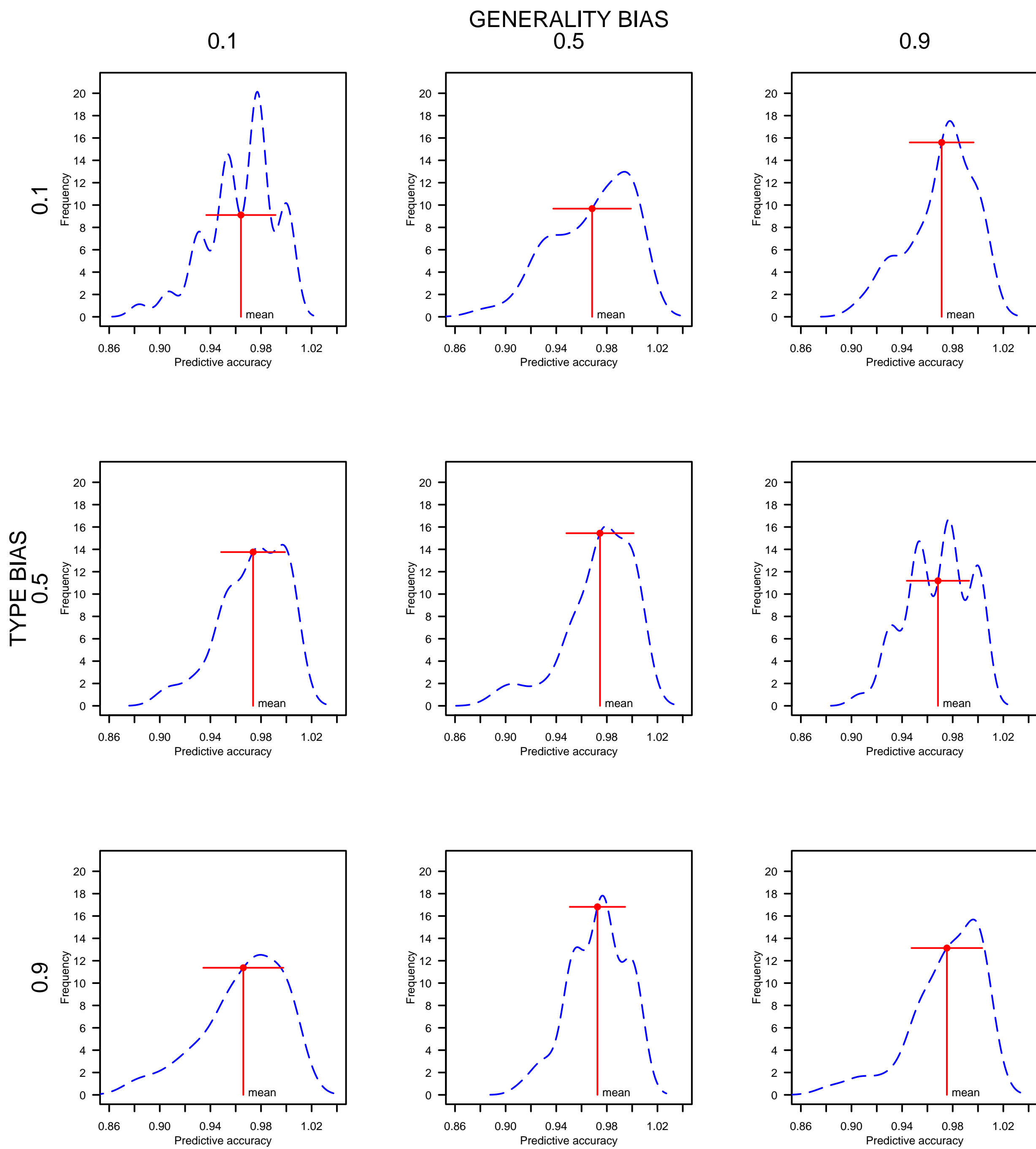


Figure 4.3: Variation in generality and detector type bias (voting data set)

of these parameters.

In addition, similar experiments were conducted, whose results are not shown, in which the rates of mutation, crossover, and creation and deletion of species were varied. The number of generations before evolution was terminated was also varied and it was found that no significant increase in classifier predictive accuracy was achieved by letting the evolutionary algorithm run for more than the 100 generations used in the experiments above. An r-contiguous bits matching algorithm was also implemented in which a detector with threshold $t \in [0, 1]$ recognised an antigen if at least $t \times b$ consecutive bits matched, where b is the total number of bits in the detector. This configuration was found to generally produce classifiers of lower predictive accuracy than the percentage-of-matching-bits scheme described above and used in the current implementation.

4.3 Classifier structure

One of the major advantages of the AIS classifier described here is the comprehensibility of the classification rules it produces. To produce a rule-based description for a classifier, the first 2-bit schema of each detector was converted to a test of the first vote, the second 2-bit schema to a test of the second vote, and so on, using the same schema interpretation of Potter and De Jong [78] and given in **Table 4.3**. **Table 4.4** gives the encoding scheme for the voting data, originally given in **Section 3.3.2** and reproduced here in order to ease interpretation of **Table 4.3**. In this table partial matches are taken into account and given a half credit, indicated by the $\frac{1}{2}$ subscript. **Tables 4.5** and **4.6** show the rule-based interpretation for the Democrat and Republican detectors respectively of the best classifier evolved after a typical 100-generation 10-fold crossvalidation run of the coevolutionary algorithm, along with the threshold values for each detector. Threshold values in the rule-based interpretation can be seen as roughly indicating the fraction of tests in the rule which need to be true in order for the rule's result to be true. This structure is similar to that of the classifiers of Potter and De Jong, and shows a graduation from detectors with a low threshold and few rules to detectors with high thresholds and a greater number of rules. The evolutionary algorithm appears to search for attribute values which will classify a large number of the examples, as in those of votes 3 and 4, and assigns low thresholds to these rules, increasing the threshold value as the complexity of the rules expressed in the detectors increases.

The number of detectors in evolved classifiers is also of interest, as it is desirable for reasons of parsimony to have classifiers with as few rules as

possible in order to increase classification speed and aid the extraction of information about the data. **Table 4.7** gives the mean number of detectors observed in the crossvalidation experiments of **Section 4.1** above. This table shows a general bias towards a greater number of Democrat rather than Republican detectors, suggesting that recognising Democrats rather than Republicans was a more fruitful strategy for the evolved classifiers. These results also compare favourably to those reported by Potter and De Jong [78] for the AQ15 system, which, with a mean size of 15.10 for the voting problem, generally produced a much larger number of cover elements.

Schema	**Interpretation**
00	abstain or yea$_{\frac{1}{2}}$ or nay$_{\frac{1}{2}}$
01	yea or abstain$_{\frac{1}{2}}$
10	nay or abstain$_{\frac{1}{2}}$
11	yea$_{\frac{1}{2}}$ or nay$_{\frac{1}{2}}$
0#	abstain or yea
1#	nay
#0	abstain or nay
#1	yea
##	ignore

Table 4.3: Schema interpretation for rule-based description

vote	**encoding**
abstain	00
yea	01
nay	10

Table 4.4: Encoding scheme for voting data set

Antibody 1	`#####110########################`
0.505882	
if	vote 3 = yea
	vote 4 = nay *or* abstain$_{\frac{1}{2}}$
then	class = democrat
Antibody 2	`#############01#########1#####1#`
0.752941	
if	vote 7 = abstain *or* yea
	vote 8 = nay
	vote 13 = nay
	vote 16 = nay
then	class = democrat
Antibody 3	`####0############1100###########`
0.756863	
if	vote 3 = abstain *or* yea
	vote 9 = yea
	vote 10 = nay *or* abstain$_{\frac{1}{2}}$
	vote 11 = abstain *or* yea
then	class = democrat
Antibody 4	`#0#10#0###0##########1##########`
0.894118	
if	vote 1 = abstain *or* yea
	vote 2 = yea
	vote 3 = abstain *or* yea
	vote 4 = abstain *or* yea
	vote 6 = abstain *or* yea
	vote 11 = nay
then	class = democrat
Antibody 5	`#1#1#0#10####1##00####0######0##`
0.913725	
if	vote 1 = yea
	vote 2 = yea
	vote 3 = abstain *or* yea
	vote 4 = yea
	vote 5 = abstain *or* yea
	vote 7 = yea
	vote 9 = abstain *or* yea$_{\frac{1}{2}}$ *or* nay$_{\frac{1}{2}}$
	vote 12 = abstain *or* yea
	vote 15 = abstain *or* yea
then	class = democrat

Table 4.5: AIS rule interpretation (Democrat detectors)

Antibody 1 0.596078	`#01###0#0##########00########1##`
if	vote 1 = abstain *or* yea
	vote 2 = nay
	vote 4 = abstain *or* yea
	vote 5 = abstain *or* yea
	vote 10 = abstain *or* yea
	vote 11 = abstain *or* yea
	vote 15 = yea
then	class = republican
Antibody 2 0.937255	`###1###10##1#####1###0#1########`
if	vote 2 = yea
	vote 4 = yea
	vote 5 = abstain *or* yea
	vote 6 = yea
	vote 9 = yea
	vote 11 = abstain *or* yea
	vote 12 = yea
then	class = republican

Table 4.6: AIS rule interpretation (Republican detectors)

	mean	**standard deviation**	**min**	**max**
Democrat	4.70	0.30	3	6
Republican	2.28	0.74	1	4
Total	6.98	0.67	4	10

Table 4.7: Number of detectors in evolved classifiers

Chapter 5

Experiments and analysis - HTML document classification

"*Experiments are mediators between nature and idea.*"

Johann Wolfgang Von Goethe (1749-1832).

The last chapter showed that the AIS concept learner was able to find good solutions to the voting problem. In this chapter, we report on the performance of the concept learner on the second of our problems, that of classifying web pages as relevant or irrelevant based on a user provided training set of ranked pages.

5.1 Classifier performance

The previous chapter used 10-fold crossvalidation to estimate the predictive accuracy of the classifiers produced by our two learning algorithms. This was acceptable as the size of the data set, 435 samples in all, was large enough to produce a reliable estimate. However, the four data sets used in this section range from between 61 to 131 samples, making crossvalidation an unreliable means of estimating the predictive accuracy due to the relatively small size of the test set produced by this method [63]. In such cases alternative methods need to be employed, one of which, and the one used here, is to randomly create a training set by randomly selecting n samples from the original data set without replacement. The remaining unselected samples then become the test set. These two sets can then be used to give a reliable estimate of the predictive accuracy for classifiers trained with a training set of size n. This method is also advantageous for our purposes in that it can be used to assess the performance of classifiers over a range of training set sizes, giving

a good indication of number of pages a user would have to rate in order to get reliable results from the classifier.

Figure 5.1 shows the predictive accuracy of the AIS and NBC learning algorithms trained using a number of different training set sizes on a feature vector with 128 features. The training set size extends along the x-axis, and the mean predictive accuracy of classifiers trained with a set of this size along the y-axis. For each training set size we randomly selected the appropriate number of training examples, using the remainder as a test set. In order to transform the HTML documents into feature vector representations, after the training and test sets were created, the 128 most informative words were extracted from the training set documents, and these words used to transform the training and test sets into Boolean feature vectors in the process described in **Section 3.5**. This process was repeated 30 times for each training set size, and the mean predictive accuracy of the resulting classifiers plotted in **Figure 5.1** for each of the four data sets: Bands, BioMedical, Goats and Sheep, described in **Section 3.3.2** above. Summaries of these results, including standard deviations and confidence intervals are given in **Appendix A** in **Tables A.1**, **A.2**, **A.3** and **A.4**.

The first thing that can be noted from the graphs in **Figure 5.1** is the somewhat lower predictive accuracy of both classifiers on this classification problem compared with that of the voting problem of the previous chapter. This is not suprising as text classification problems are generally considered to be relatively hard classification problems [63], and in an informal survey of the literature on text classification we found the best classifiers to be achieving predictive accuracies of around 0.70 whatever the text classification problem. In this context, the performance of both classifiers is more than reasonable on the four problems. The results presented for the NBC are also similar to those of the NBC-based system of Pazzani et al. [72], offering an indication that the performance levels are due to the nature of the problem and not a result of implementation problems.

This said, there are marked differences in the predictive accuracies of the NBC and AIS classifiers on the four problems, with the AIS algorithm at first sight appearing to generally perform better on all data sets, except the bands set, than the naive Bayesian classifier. As before, a Wilcoxon rank sum significance test was performed on the results and this confirmed that there was a statistically significant difference between the predictive accuracies of the two algorithms, except on the bands data set. From this we can conclude that the AIS classifier in general consistently produces better classifiers than those produced by the NBC algorithm.

Also interesting to note is the relatively constant performance of the AIS classifiers over a range of training set sizes. Increasing the size of the training

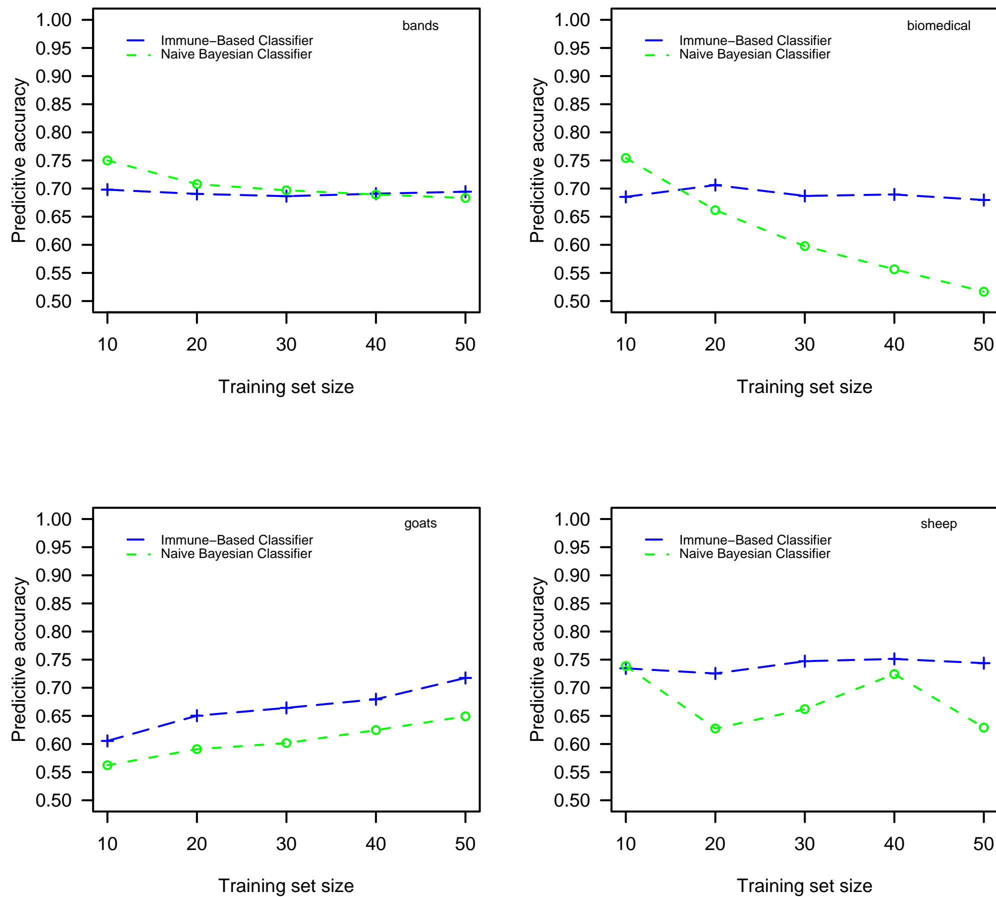


Figure 5.1: Classifier performance (document classification)

set seemed to produced little increase in classifier performance for this algorithm, while for the NBC there was a much greater fluctuation in classifier performance on an increase in training set size. This constancy of AIS performance is particularly useful on problems such as this one because classifiers which perform well on a small training set size would be advantageous as users would be able to rate less pages but still obtain accurate predictions.

5.2 Evolutionary algorithm dynamics

Similar experiments to those described in the **Section 4.2** above were performed to obtain a picture of the dynamics of the AIS algorithm over a typical evolutionary run, and to assess the role of various parameter setting on algorithm performance. These results were on the whole similar to those previously presented in **Section 4.2** for the voting problem and are not given here but instead relegated to **Appendix A** where they are included for completeness. Since we are particularly interested in the performance of the AIS classifier on relatively small training sets, the graphs given in the appendix were all obtained using a training set size of 20 on the sheep data set. **Figure A.1** shows the dynamics of a standard evolutionary run as in **Figure 4.2** above, and **Figure A.2** the consequences of varying the generality and detector biases as in **Figure 4.3** above. The same conclusions as were reached in **Section 4.2** concerning the variation of these parameters, namely that the evolutionary algorithm is generally robust to changes in their values, can on reviewing these figures be seen to also apply to the classification problem of this chapter.

5.3 Feature extractor dynamics

The feature extraction algorithm determines the number of words which are to be represented in the feature vector and thus the amount of information the classifier has available to classify a particular sample. In order to study the effects changes in feature vector length have on classifier performance we varied the number of words used as features, the results of which are shown in **Figures 5.2** and **5.3** for the AIS and NBC respectively. These results were obtained on the sheep data set for a range of training set sizes, represented along the x-axis, using the same method as that of the trials described previously in **Section 5.1**.

Figure 5.2 shows the AIS constantly producing classifiers with a predictive accuracy of around 0.75 for features vectors of any length, while in

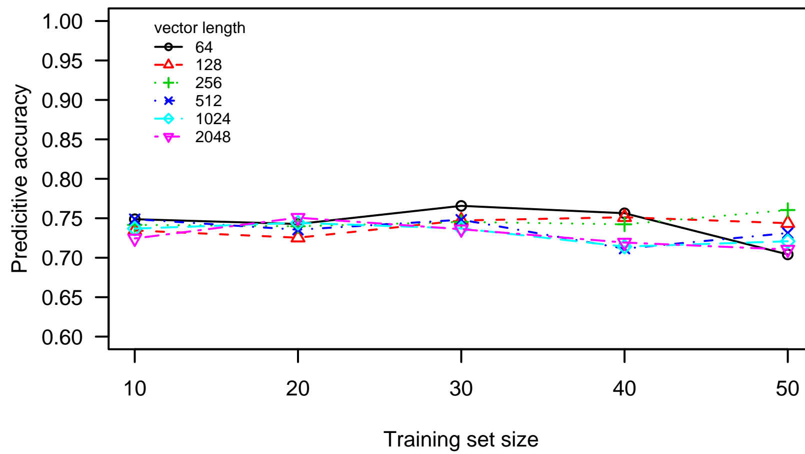


Figure 5.2: Variation in feature vector length (AIS)

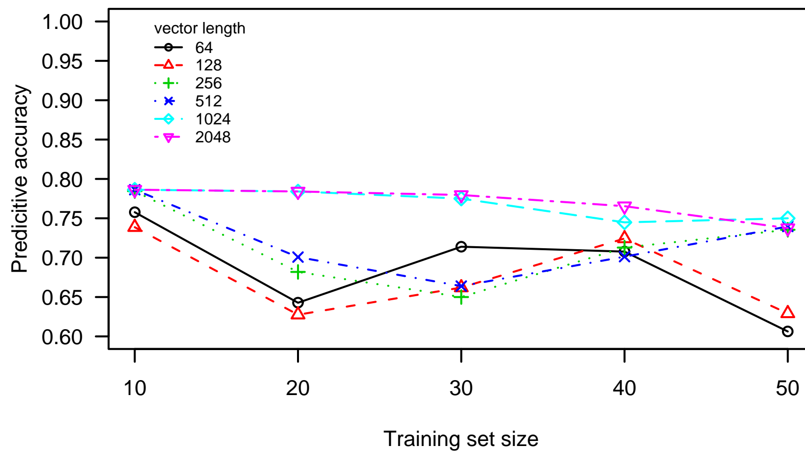


Figure 5.3: Variation in feature vector length (NBC)

Figure 5.3 the NBC generally produces classifiers with a lower accuracy. The exception is for feature vectors with 1024 and 2048 features, where the predictive accuracy for the NBC is higher than that of the AIS. Increasing the number of features in the feature vector can have two effects. It provides the classifier with more information, so predictive accuracy should rise, but it also introduces more noise, making the classifier's job harder. Also, in practical systems, long feature vectors are disadvantageous, as they require large amounts of storage space and more processing time. The advantage of the AIS concept learner here is that it is able to consistently produce classifiers with better predictive accuracies from feature vectors with few features than the NBC.

Chapter 6

Discussion

> *"Twice and thrice over, as they say, good is it to repeat and review what is good."*
>
> Plato (c. 427-347 B.C.)

In this chapter we review and discuss the results presented in the last two chapters, considering the performance of the AIS classifier across the two problem domains on which it was evaluated. We begin by discussing reasons for the differences in performance seen over these two domains.

While the AIS classifier outperforms all the learning algorithms it has been compared against in **Chapters 4** and **5**, there are marked differences in the levels of predictive accuracies it achieves on the voting and document classification problems. One factor which can be attributed to contributing to these differences is the variation in the size of the training sets available for each problem. In terms of available data, the voting problem represents a fairly data-rich classification problem, with 435 samples available, whereas the number of available samples for the document classification problem, between 61 and 131, makes it a relatively data-sparse problem. This leads to less samples being available for classifier training, and so it is expected that performance will generally be lower on the data-sparse problem [63].

A second factor contributing to the difference in performance across the two problems is the feature extraction algorithm. In this dissertation we use a statistical feature extraction algorithm which extracts features based on their expected information gain and, while necessary to provide a principled comparison with the NBC-based system of Pazzani et al. [72], as discussed in **Section 2.2.1** such a method is far from ideal in the context of HTML document classification. The coarse-grained document representation produced by our feature extractor introduces a fair amount of noise into the system, making classification generally that much harder than in the voting prob-

lem. Using a more fine-grained feature extractor, such as those described by Cohen [13] or Yang et al. [100], which also exploits meta-features of HTML documents such as hyperlinks and tags, would potentially increase classifier performance on the document classification problem and close the gap in performance difference. Other potential ways of increasing the performance of our AIS classifier are discussed in the next chapter.

An issue which has not been raised so far is that of classifier training time. In practical terms, choice of a classifier not only depends upon the predictive accuracies it is able to achieve, but also on the amount of time taken in training the classifier. No matter how good the results achieved, users would often be unwilling to wait for more than a few seconds for these results, and definitely not, for example, the several days or even months taken by some of the algorithms reported by Lim et al. [60]. While our immune-based classifier is obviously more computationally expensive than the naive Bayesian classifier, for the voting problem of **Chapter 4**, a typical 100-generation run with 400 training examples took around 40 seconds. For the document classification problem of **Chapter 5**, the immune-based classifier took around 6 seconds to train over 100 generations on a training set of size 20 with 128 features. These figures are of course to some extent implementation and system specific, further details of which are given in **Appendix B**, but nevertheless show that as well as achieving better predictive accuracies than that of many other classifiers, our immune-based classifier is able to do so in a time which allows it to be applied to real-world problems.

In summary, **Chapters 4** and **5** show that we have been able to achieve the aim of this project, namely: we have produced a novel, working system built on an immune-based learning algorithm which is able to perform better than the currently available learning algorithms. In order to give substance to this claim, we have compared our system's performance to that of other systems in a systematic and rigorous manner. We now move on to discuss possible directions for future research.

Chapter 7

Future work

> *"All progress is precarious, and the solution of one problem brings us face to face with another problem."*
>
> Martin Luther King Jr. (1929-1968).

This section explores several possibilities for further work that could build on the work presented here but which, due to time and space constraints, we were unable to include in the current implementation. First, we consider several potential applications of the present system not explored here, and then go on to consider a number of changes to the current system which could further enhance its performance.

Many potential document classification problems are dynamic in nature, meaning that the data on which the concept learner bases its hypotheses changes over time. Take for example a system in which users have a collection of documents from which the concepts of '*related*' and '*unrelated*' are learnt. Over time, users may add or remove documents to and from this collection. Instead of relearning the concepts from scratch each time a document is added, as is necessary with naive Bayesian classifier, it would be interesting to see if the AIS concept learner was able to produce accurate hypotheses starting from the previously learned concept, and so potentially offer savings in the amount of training time necessary. This could be implemented by keeping the final species from the evolutionary run which produced the last concept and starting the evolution of the new species from these individuals, instead of from the one random species currently used to begin evolutionary runs. Another possibility which would involve less storage overheads would be to initialise individuals as mutations of the current detector set. Gaspar and Collard [37] study the performance of an immune-based system on a time-dependent optimisation problem and find that it performs well against a standard genetic algorithm, a performance which they attribute to the

central role of diversity within the adaptive dynamics of their system, giving a further indication that immune-based systems may also be advantageous in dynamic classification problems.

Another little-explored but potentially useful study would be to evaluate the performance of the AIS learning algorithm at the task of learning from positive examples only. One can imagine a system in which users, instead of having to formulate a standard '*string of words*' search query to input into a web search engine, instead simply highlight a number of documents and ask the system to find related documents. Since, in this example, no negative i.e. irrelevant examples are provided, the system would have to construct its hypotheses purely from positive evidence. This is essentially how mature T-cells in the human immune system are produced in that, during negative selection, immature T-cells are only exposed to self proteins, and so would suggest, along with the previously mentioned study of Hunt and Cooke [49], that a system employing similar dynamics might also be successful at this task.

The possibility is currently being explored of implementing the AIS concept learner described here in a real-world application, the ePerson collaborative information management system being developed at Hewlett-Packard Laboratories, Bristol. This system is a platform in which new paradigms in information management and retrieval are being explored. As part of its functionality, a workspace is provided in which users can place documents and organise them in a bookmark-style hierarchy, with different users choosing different categorisation schemes. For example, consider two researchers collaborating on solving a problem. Both researchers may have compiled collections of documents related to the problem, but organised them in different ways suiting their style of working. The AIS concept learner, by taking the documents in one category as positive examples, and all the documents in all the other categories as negative examples, is able to learn a concept for the category. This concept can then be used to determine if any of the documents from the second user's workspace fit into this category. By repeating this for all the categories in the first user's workspace, the documents in the second user's workspace can be automatically classified according to the first user's categorisation scheme.

An area of research which has attracted increasing interest in recent years within the machine learning community is that of ensemble learning [24]. As the name suggests, an ensemble is a collection of individually trained classifiers who decisions are combined in some manner, such as boosting [6] or bagging [36], and is seen as a way of improving classifier performance, with the performance of an ensemble of classifiers often found to be higher than that of its individual constituent classifiers [67]. It would be interesting to ex-

plore this possibility with an immune-based classifier to see if any additional performance increases could be obtained.

The algorithms used in the implementation of the immune concept learning system explored in this dissertation constrain the form and general properties of the classifiers it produces. Changing these algorithms could potentially enhance the performance of the system, and while several different possibilities such as changing the matching algorithm were explored, as outlined in **Section 4.2**, a number of further possibilities exist. One such possibility, that of using a more fine-grained feature extraction algorithm, has already been mentioned in the previous chapter. In terms of the mechanisms at work in the human immune system those of the artificial immune system described in this dissertation are, to say the least, simplistic, and present a very crude analogy to their biological counterparts. Nevertheless, even from such humble an analogy it has been shown that a powerful concept learning system can be created, and perhaps with increased fidelity to its biological counterpart, further increases in performance can be gained. One such possibility involves a more realistic implementation of the processes of humoral immunity as described in **Section 2.2** and pictured in **Figure 2.2**. Instead of the simple T-cell driven model of the current system, implementation of both B- and T-cells and their associated dynamics in the human immune system, such as in the systems of Carter [7] or Cayzer and Aickelin [8, 9] described in **Section 2.3.2.1**, could permit a more accurate classification of documents through processes akin to affinity maturation and clonal selection.

Chapter 8

Conclusions

> "*There are so many ways to wear what we have before it's gone.*"
>
> Ani Di Franco (1970-).

We have reached the end of the journey which this dissertation represents and which we now briefly review. It began in **Chapter 1** with the laying out in broad terms of the general themes and concepts on which the work presented here rested, and of the aims and motivation behind this work. **Chapter 2** presented and discussed in detail the key concepts and work related to the immune-based system we implemented, which itself was elaborated in **Chapter 3**, along with the other systems we implemented and the data we used to provide a principled comparison to other related systems. The performance and dynamics of our immune-based system were analysed and contrasted with that of other concept learners on a standard classification task in **Chapter 4**, where we found the performance of the immune-based system to be significantly better than its contemporaries. In **Chapter 5** we then compared our immune-based learner with other learning algorithms on a web-based document classification task, and also found that it performed significantly better. Conclusions on this performance across both tasks were drawn and discussed in **Chapter 6**, which led on to the suggestions for future work presented in **Chapter 7**.

My final task is to review the aim of the dissertation as originally set out in **Chapter 1** and to assess if I have achieved this aim. My aim was to produce a novel, working system built on a immune-based learning algorithm which was able to perform better than the currently available learning algorithms on a standard classification problem and a web-based document classification problem. As detailed in the previous paragraph, the immune-based system was able to consistently outperform the systems with which it was compared,

and as such, achieved its aim.

Bibliography

[1] V. Atluri, C. Hung, and T. L. Coleman. Artificial immune systems for soil data classification. In S. Y. Shin, editor, *Proc. of the Fifteenth Int. Conf. on Computers and their Applications (CATA-2000)*, pages 358–360. ICSA Press, New Orleans, LA, 2000.

[2] J. A. Bellanti and J. V. Kadlec. Introduction to immunology. In J. A. Bellanti, editor, *Immunology: Basic Processes*, chapter 1, pages 1–15. W.B. Saunders Company, Philadelphia, PA, 1985.

[3] D. Billsus and M. Pazzani. A hybrid user model for news story classification. In J. Kay, editor, *Proc. of the Seventh Int. Conf. on User Modeling*, pages 99–108. Springer-Verlag, Banff, Canada, 1999.

[4] C. L. Blake and C. J. Merz. UCI Repository of Machine Learning Databases. Department of Information and Computer Sciences, University of California, Irvine, CA. *http://www.ics.uci.edu/~mlearn/MLRepository.html*.

[5] H. Blockeel, W. Van Laer, and L. De Raedt. Inductive constraint logic and the mutagenesis problem. In J. C. Meyer and L. C. Van Der Gaag, editors, *Proc. of the Eighth Dutch Conf. on Artificial Intelligence*, pages 265–276. Universiteit Utrecht, Utrecht, 1996.

[6] L. Breiman. Stacked regressions. *Machine Learning*, 24:41–48, 1996.

[7] J. H. Carter. The immune system as a model for pattern recognition and classification. *Journal of the American Medical Informatics Association*, 7(1):28–41, 2000.

[8] S. Cayzer and U. Aickelin. On the effects of idiotypic interactions for recommendation communities in artificial immune systems. In J. Timmis and P. Bentley, editors, *Proc. of the First Int. Conf. on Artificial Immune Systems (ICARIS-2002)*. Canterbury, U.K., 2002 (*forthcoming*).

[9] S. Cayzer and U. Aickelin. A recommender system based on the immune network. In P. Fogel, editor, *Congress on Evolutionary Computation 2002 (CEC-2002)*. IEEE Press, Honolulu, Hawaii, 2002 (*forthcoming*).

[10] F. Ciravegna. $(LP)^2$, an adaptive algorithm for information extraction from web-related texts. In B. Nebel, editor, *Proc. of the Workshop on Adaptive Text Extraction and Mining (IJCAI-2001)*, pages 66–76. Morgan-Kaufmann, Seattle, WA, 2001.

[11] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[12] W. W. Cohen. Recognizing structure in web pages using similarity queries. In James H. and D. Subramanian, editors, *Proc. of the Sixteenth Amer. Nat. Conf. on Artificial Intelligence (AAAI-99)*, pages 59–66. AAAI Press, Orlando, FL, 1999.

[13] W. W. Cohen. Automatically extracting features for concept learning from the web. In P. Langley, editor, *Proc. of the Seventeenth Int. Conf. on Machine Learning (ICML-2000)*, pages 159–166. Morgan-Kaufmann, San Francisco, CA, 2000.

[14] J. P. Cohoon, S. U. Hegde, W. N. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. In J. J. Grefenstette, editor, *Proc. of the Second Int. Conf. on Genetic Algorithms*, pages 148–154. Lawrence Erlbaum Assoc., MIT, Cambridge, 1987.

[15] A. Coutinho. Beyond clonal selection and network. *Immunol. Rev.*, 110:63–87, 1989.

[16] D. Dasgupta. Information processing mechanisms of the immune system. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, chapter 3, pages 45–67. McGraw-Hill, New York, 1999.

[17] D. Dasgupta and N. Attoh-Okine. Immunity-based systems: a survey. In *Proc. of the IEEE Int. Conf. on Systems, Man and Cybernetics*, volume 1, pages 369–374. IEEE Press, Orlando, FL, 1997.

[18] D. Dasgupta and F. A. González. Evolving complex fuzzy classifier rules using a linear genetic representation. In L. Spector, D. Whitley, D. Goldberg, E. Cantu-Paz, I. Parmee, and H. Beyer, editors, *Proc. of the Int. Conf. on Genetic and Evolutionary Computation (GECCO-2001)*, pages 299–305. Morgan-Kaufmann, San Francisco, CA, 2001.

[19] D. Dasgupta, N. Majumdar, and F. Nino. Artificial immune systems: a bibliography. Technical Report CS-01-002 Version 2.0, Computer Science Division, The University of Memphis, Memphis, TN, 2001.

[20] L. N. De Castro and J. Timmis. Artificial immune systems: a novel approach to pattern recognition. In L. Alonso, J. Corchado, and C. Fyfe, editors, *Artificial Neural Networks in Pattern Recognition*, pages 67–84. University of Paisley, U.K., 2002.

[21] L. N. De Castro and F. J. Von Zuben. Artificial immune systems: part II - a survey of applications. Technical Report DCA-RT 02/00, Department of Computer Engineering and Industrial Automation, School of Electrical and Computer Engineering, State University of Campinas, SP, Brazil, 2000.

[22] K. A. De Jong, W. M. Spears, and D. F. Gordon. Using genetic algorithms for concept learning. *Machine Learning*, 13:161–188, 1993.

[23] K. Deb. Genetic algorithms in multimodal function optimization. Master's thesis, Department of Engineering Mechanics, The University of Alabama, Tuscaloosa, AL, 1989.

[24] T. G. Dietterich. Machine-learning research: four current directions. *The AI Magazine*, 18(4):97–136, 1998.

[25] R. Duda and P. Hart. *Pattern classification and scene analysis.* John Wiley & Sons, New York, 1973.

[26] T. Eliassi-Rad and J. Shavlik. Instructable and adaptive web-agents that learn to retrieve and extract information. Technical Report 2000-1, Machine Learning Research Group, Department of Computer Sciences, University of Wisconsin, 2000.

[27] S. Endoh, N. Toma, and K. Yamada. Immune algorithm for n-TSP. In *Proc. of the IEEE Int. Conf. on Systems, Man and Cybernetics*, volume 4, pages 3844–3849. IEEE Press, San Diego, CA, 1998.

[28] J. D. Farmer, N. H. Packard, and A. S. Perelson. The immune system, adaptation and machine learning. *Physica D*, 22:187–204, 1986.

[29] G. Folino, C. Pizzuti, and G. Spezzano. A cellular genetic programming approach to classification. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proc. of the Genetic and Evolutionary Computation Conf.*, volume 2, pages 1015–1020. Morgan-Kaufmann, Orlando, FL, 1999.

[30] S. Forrest and S. A. Hofmeyr. John Holland's invisible hand: an artificial immune system. Presented at the Festschrift held in honor of John Holland, 1999.

[31] S. Forrest and S. A. Hofmeyr. Immunology as information processing. In L. A. Segal and I. R. Cohen, editors, *Design Principles for Immune Systems and Other Distributed Autonomous Systems*, chapter 17, pages 361–388. Oxford University Press, New York, 2001.

[32] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, 1997.

[33] S. Forrest, B. Javornik, R. E. Smith, and A. S. Perelson. Using genetic algorithms to explore pattern recognition in the immune system. *Evolutionary Computation*, 1(3):191–211, 1993.

[34] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. A change-detection algorithm inspired by the immune system. Submitted to IEEE Transactions on Software Engineering, 1995.

[35] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *Proc. of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 202–212. IEEE Press, Los Alamitos, CA, 1994.

[36] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In L. Saitta, editor, *Proc. of the Thirteenth Int. Conf. on Machine Learning*, pages 148–156. Morgan-Kaufmann, San Francisco, CA, 1996.

[37] A. Gaspar and P. Collard. From GAs to artificial immune systems: improving adaptation in time dependent optimization. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proc. of the Congress on Evolutionary Computation*, volume 3, pages 1859–1866. IEEE Press, Mayflower Hotel, Washington D.C., 1999.

[38] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[39] R. A. Goldsby, T. J. Kindt, and B. A. Osborne. *Kuby Immunology.* W. H. Freeman and Company, New York, 4th edition, 2000.

[40] E. Hart and P. Ross. Clustering moving data with a modified immune system. In E. J. W. Boers, S. Cagnoni, J. Gottlieb, E. Hart, P. L.

Lanzi, G. Raidl, R. E. Smith, and H. Tijink, editors, *Applications of Evolutionary Computing, EvoWorkshop 2001*, pages 394–403. Springer-Verlag, Berlin, 2001.

[41] S. A. Hofmeyr. *An Immunological Model of Distributed Detection and its Application to Computer Security.* PhD thesis, Department of Computer Science, University of New Mexico, Albuquerque, NM, 1999.

[42] S. A. Hofmeyr. An interpretative introduction to the immune system. In L. A. Segal and I. R. Cohen, editors, *Design Principles for Immune Systems and Other Distributed Autonomous Systems*, chapter 1, pages 3–28. Oxford University Press, New York, 2001.

[43] S. A. Hofmeyr and S. Forrest. Immunity by design: an artificial immune system. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proc. of the Genetic and Evolutionary Computation Conf.*, volume 2, pages 1289–1296. Morgan-Kaufmann, Orlando, FL, 1999.

[44] S. A. Hofmeyr and S. Forrest. Architecture for an artificial immune system. *Evolutionary Computation*, 8(4):443–473, 2000.

[45] J. H. Holland. *Adaptation In Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1976.

[46] J. H. Holland. Escaping brittleness: the possibilities of general purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2, pages 593–623. Morgan-Kaufmann, Los Altos, CA, 1986.

[47] J. H. Holland, K. J. Holyoak, R. E. Nisbett, and P. Thagard. *Induction: Processes of Inference, Learning, and Discovery.* MIT Press, Cambridge, MA, 1986.

[48] S. Huang. An immune-based optimization method for capacitor placement in radial distribution system. *IEEE Transactions on Power Delivery*, 15(2):744–749, 2000.

[49] J. E. Hunt and D. E. Cooke. Learning using an artificial immune system. *Journal of Network and Computer Applications*, 19:189–212, 1996.

[50] J. E. Hunt, C. King, and D. E. Cooke. Immunizing against fraud. In *Proc. of the IEE Colloquium on Knowledge Discovery and Data Mining*, volume 4, pages 1–4. IEE Press, U.K., 1996.

[51] R. Ihaka and R. Gentleman. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[52] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[53] N. K. Jerne. The immune system. *Scientific American*, 229(1):52–60, 1973.

[54] N. K. Jerne. Towards a network theory of the immune system. *Ann. Immunol. (Inst. Pasteur)*, 125C:373–379, 1974.

[55] L. Jiao and L. Wang. A novel genetic algorithm based on immunity. In *IEEE Transactions on Systems, Man and Cybernetics*, volume 30, pages 552–561. IEEE Press, San Diego, CA, 2000.

[56] P. Kanerva. *Sparse distributed memory.* MIT Press, Cambridge, MA, 1988.

[57] J. R. Koza. Genetic programming. In J. G. Williams and A. Kent, editors, *Encyclopedia of Computer Science and Technology*, volume 39, pages 29–43. Marcel-Dekker, New York, 1998.

[58] F. W. Lancaster. *Information Retrieval Systems: Characteristics, Testing and Evaluation.* John Wiley and Sons, New York, 2nd edition, 1979.

[59] K. Lang. NewsWeeder: learning to filter news. In A. Prieditis and S. Russell, editors, *Proc. of the Twelfth Int. Conf. on Machine Learning (ICML-95)*, pages 331–339. Morgan-Kaufmann, Lake Tahoe, CA, 1995.

[60] T. Lim, W. Loh, and Y. Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine Learning*, 40(3):203–228, 2000.

[61] R. S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20(2):111–161, 1983.

[62] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, 1996.

[63] T. M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[64] K. Mori, M. Tsukiyama, and T. Fukuda. Adaptive scheduling systems inspired by immune system. In *Proc. of the IEEE Int. Conf. on Systems, Man and Cybernetics*, volume 4, pages 3833–3837. IEEE Press, San Diego, CA, 1998.

[65] T. Morrison and U. Aickelin. An artificial immune system as a recommender system for web sites. In J. Timmis and P. Bentley, editors, *Proc. of the First Int. Conf. on Artificial Immune Systems (ICARIS-2002).* Canterbury, U.K., 2002 (*forthcoming*).

[66] F. Neri. A study on the effect of cooperative evolution on concept learning. In E. J. W. Boers, S. Cagnoni, J. Gottlieb, E. Hart, P. L. Lanzi, G. Raidl, R. E. Smith, and H. Tijink, editors, *Applications of Evolutionary Computing, EvoWorkshop 2001*, pages 314–320. Springer-Verlag, Berlin, 2001.

[67] D. Opitz and R. Maclin. Popular ensemble methods: an empirical study. *Journal of Artificial Intelligence Research*, 11:169–198, 1999.

[68] D. G. Osmond. The turn-over of B-cell populations. *Immunol. Today*, 14(1):34–37, 1993.

[69] G. Pant and F. Menczer. MySpiders: evolve your own intelligent web crawlers. *Autonomous Agents and Multi-Agent Systems*, 5(2):221–229, 2002.

[70] M. Pazzani. Syskill and Webert web page ratings. UCI Repository of Machine Learning Databases, Department of Information and Computer Sciences, University of California, Irvine, CA. *http://www.ics.uci.edu/~mlearn/MLRepository.html.*

[71] M. Pazzani and D. Billsus. Learning and revising user profiles: the identification of interesting web sites. *Machine Learning*, 27:313–331, 1997.

[72] M. Pazzani, J. Muramatsu, and D. Billsus. Syskill and Webert: identifying interesting websites. In W. J. Clancey and D. Weld, editors, *Proc. of the Thirteenth Amer. Nat. Conf. on Artificial Intelligence (AAAI-96)*, volume 1, pages 54–61. AAAI Press, Portland, OR, 1996.

[73] C. C. Pettey. Diffusion (cellular) models. In T. Bäck, D. B. Fogel, and Z. Michalewicz, editors, *The Handbook of Evolutionary Computation*, pages 24–33. Oxford University Press, New York, 1997.

[74] H. Pospesel. *Introduction to Logic: Propositional Logic.* Prentice Hall, New York, 3rd edition, 1997.

[75] M. A. Potter and K. A. De Jong. A cooperative coevolutionary approach to function optimization. In Y. Davidor, H. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature – PPSN-94*, pages 249–257. Springer-Verlag, Berlin, 1994.

[76] M. A. Potter and K. A. De Jong. Cooperative coevolution: an architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.

[77] M. A. Potter, K. A. De Jong, and J. J. Grefenstette. A coevolutionary approach to learning sequential decision rules. In L. Eshelman, editor, *Proc. of the Sixth Int. Conf. on Genetic Algorithms*, pages 366–372. Morgan-Kaufmann, San Francisco, CA, 1995.

[78] M. A. Potter and K. A. De Jong. The coevolution of antibodies for concept learning. In A. E. Eiben, T. Bäck, M. Schoenauer, and H. Schwefel, editors, *Proc. of the Fifth Int. Conf. on Parallel Problem Solving from Nature (PPSN-98)*, pages 530–539. Springer-Verlag, Amsterdam, 1998.

[79] J. R. Quinlan. *Discovering Rules from Large Collections of Examples.* Edinburgh University Press, U.K., 1979.

[80] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[81] J. R. Quinlan. Generating production rules from decision trees. In J. McDermott, editor, *Proc. of the Tenth Int. Joint Conf. on Artificial Intelligence*, pages 304–307. Morgan-Kaufmann, San Mateo, CA, 1987.

[82] I. Rechenberg. Cybernetic solution path of an experimental problem. Library Translation No. 1122, Royal Aircraft Establishment, Farnborough, Hants., U.K., 1965.

[83] C. D. Rosin and R. K. Belew. Methods for competitive co-evolution: finding opponents worth beating. In L. Eshelman, editor, *Proc. of the Sixth Int. Conf. on Genetic Algorithms*, pages 373–380. Morgan-Kaufmann, San Francisco, CA, 1995.

[84] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representation by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, pages 318–362. MIT Press, Cambridge, MA, 1986.

[85] J. Schlimmer. 1984 United States Congressional voting records database. UCI Repository of Machine Learning Databases, Department of Information and Computer Sciences, University of California, Irvine, CA. *http://www.ics.uci.edu/~mlearn/MLRepository.html*.

[86] I. Schwab, W. Pohl, and I. Koychev. Learning to recommend from positive evidence. In D. Riecken, D. Benyon, and H. Lieberman, editors, *Proc. of the Fifth Int. Conf. on Intelligent User Interfaces 2000*, pages 241–247. ACM Press, New Orleans, LA, 2000.

[87] R. E. Smith, S. Forrest, and A. S. Perelson. Population diversity in an immune system model: implications for genetic search. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 153–165. Morgan-Kaufmann, San Mateo, CA, 1993.

[88] R. E. Smith, S. Forrest, and A. S. Perelson. Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary Computation*, 1(2):127–149, 1993.

[89] D. Sofge, K. A. De Jong, and A. Schultz. A blended population approach to cooperative coevolution for decomposition of complex problems. In P. Fogel, editor, *Congress on Evolutionary Computation 2002 (CEC-2002)*. IEEE Press, Honolulu, Hawaii, 2002 (*forthcoming*).

[90] A. Somayaji, S. A. Hofmeyr, and S. Forrest. Principles of a computer immune system. In T. Haigh, B. Blakley, M. E. Zurbo, and C. Meodaws, editors, *Meeting on New Security Paradigms 1997*, pages 75–82. ACM Press, Langdale, U.K., 1998.

[91] K. C. Tan, A. Tay, T. H. Lee, and C. M. Heng. Mining multiple comprehensible classification rules using genetic programming. In P. Fogel, editor, *Congress on Evolutionary Computation 2002 (CEC-2002)*. IEEE Press, Honolulu, Hawaii, 2002 (*forthcoming*).

[92] S. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. A. De Jong, S. Dzeroski, S. E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R. S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek, and J. Zhang. The MONK's problems: a performance comparison of different learning algorithms. Technical Report CMU-CS-91-197, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, 1991.

[93] J. Timmis and M. Neal. A resource limited artificial immune system for data analysis. *Knowledge-Based Systems*, 14:121–130, 2001.

[94] J. Timmis, M. Neal, and J. E. Hunt. Data analysis using artificial immune systems, cluster analysis and Kohonen networks: some comparisons. In *Proc. of the IEEE Int. Conf. on Systems, Man and Cybernetics*, volume 3, pages 922–927. IEEE Press, Tokyo, Japan, 1999.

[95] J. Timmis, M. Neal, and J. E. Hunt. An artificial immune system for data analysis. *Biosystems*, 5(1/3):143–150, 2000.

[96] S. Tonegawa. Somatic generation of antibody diversity. *Nature*, 302:575–581, 1983.

[97] C. J. Van Rijsbergen. *Information Retrieval.* Department of Computer Science, University of Glasgow, U.K., 2nd edition, 1979.

[98] F. Varela and A. Coutinho. Second generation immune networks. *Immunol. Today*, 12:159–167, 1991.

[99] S. M. Weiss and C. A. Kulikowski. *Computer systems that learn.* Morgan-Kaufmann, San Francisco, CA, 1991.

[100] Y. Yang, S. Slattery, and R. Ghani. A study of approaches to hypertext categorization. *Journal of Intelligent Information Systems*, 18(2):219–241, 2002.

[101] B. Zhang and Y. Seo. Personalized web-document filtering using reinforcement learning. *Applied Artificial Intelligence*, 15(7):665–685, 2001.

Appendix A

Additional results

In this appendix additional results from **Chapter 5** which would not have fitted comfortably into the main body of text are presented. **Tables A.1**, **A.2**, **A.3** and **A.4** give summaries of the results of the experiments described in **Section 5.1**, and **Figure A.1** shows the dynamics of a standard evolutionary run for these experiments. **Figure A.2**, as with **Figure 4.2**, shows the effects variations in the generality bias and type bias parameters have on classifier performance on the document classification task of **Chapter 5**.

training set size	**learning algorithm**	**predictive accuracy**	**standard deviation**	**95% confidence interval**
10	AIS	0.698	0.056	0.072
	NBC	0.750	0.000	0.030
20	AIS	0.690	0.047	0.047
	NBC	0.707	0.067	0.012
30	AIS	0.686	0.074	0.051
	NBC	0.696	0.084	0.030
40	AIS	0.690	0.081	0.034
	NBC	0.689	0.056	0.037
50	AIS	0.694	0.116	0.046
	NBC	0.683	0.105	0.068

Table A.1: bands data set performance summary

training set size	**learning algorithm**	**predictive accuracy**	**standard deviation**	**95% confidence interval**
10	AIS	0.685	0.080	0.099
	NBC	0.754	0.002	0.039
20	AIS	0.706	0.047	0.005
	NBC	0.661	0.095	0.083
30	AIS	0.686	0.056	0.034
	NBC	0.597	0.137	0.144
40	AIS	0.689	0.048	0.072
	NBC	0.556	0.155	0.193
50	AIS	0.679	0.070	0.106
	NBC	0.516	0.137	0.219

Table A.2: biomedical data set performance summary

training set size	**learning algorithm**	**predictive accuracy**	**standard deviation**	**95% confidence interval**
10	AIS	0.605	0.065	0.011
	NBC	0.562	0.055	0.074
20	AIS	0.650	0.083	0.021
	NBC	0.590	0.061	0.097
30	AIS	0.664	0.054	0.031
	NBC	0.601	0.066	0.094
40	AIS	0.679	0.087	0.014
	NBC	0.624	0.066	0.095
50	AIS	0.717	0.095	0.024
	NBC	0.649	0.073	0.112

Table A.3: goats data set performance summary

training set size	**learning algorithm**	**predictive accuracy**	**standard deviation**	**95% confidence interval**
10	AIS	0.734	0.070	0.060
	NBC	0.738	0.134	0.051
20	AIS	0.725	0.062	0.040
	NBC	0.627	0.142	0.155
30	AIS	0.747	0.049	0.045
	NBC	0.662	0.097	0.125
40	AIS	0.751	0.067	0.018
	NBC	0.724	0.104	0.072
50	AIS	0.743	0.106	0.045
	NBC	0.629	0.153	0.183

Table A.4: sheep data set performance summary

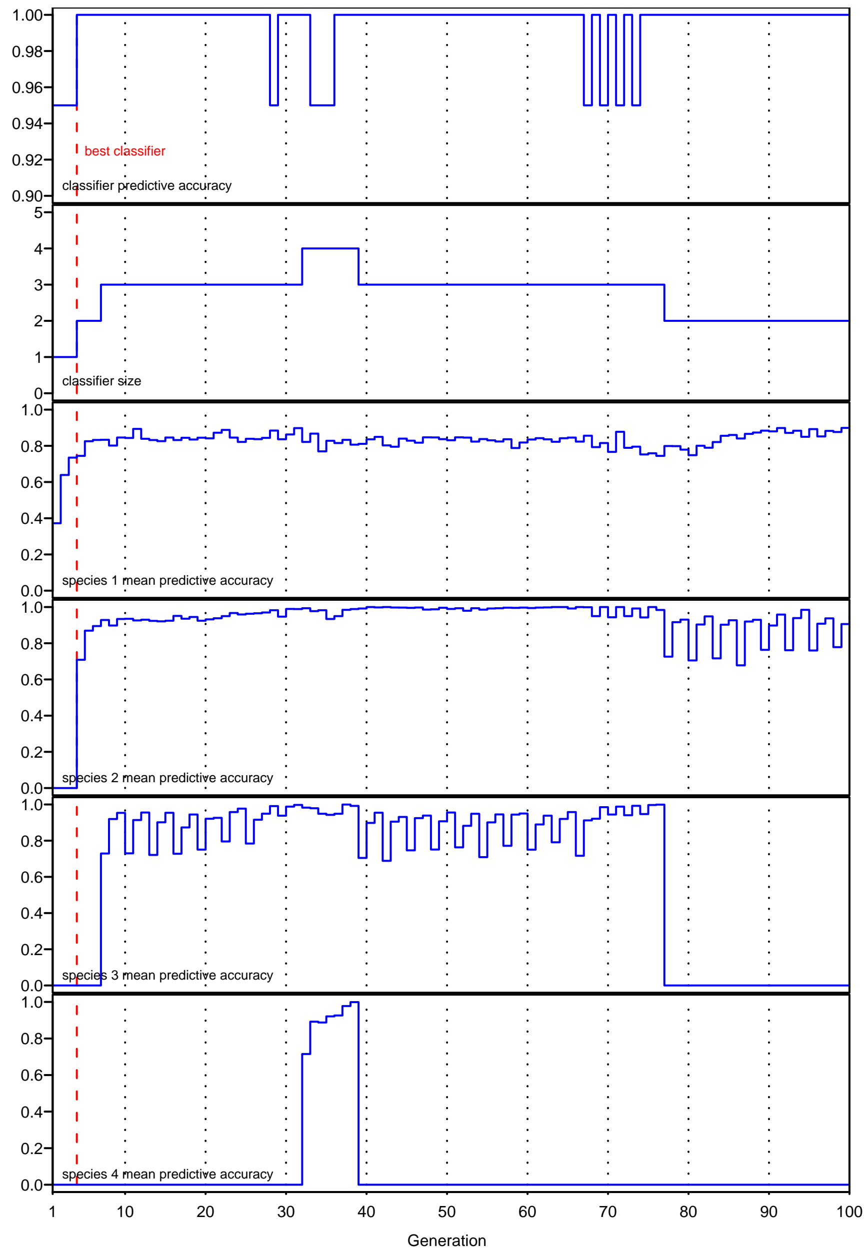


Figure A.1: Evolutionary algorithm dynamics (document classification)

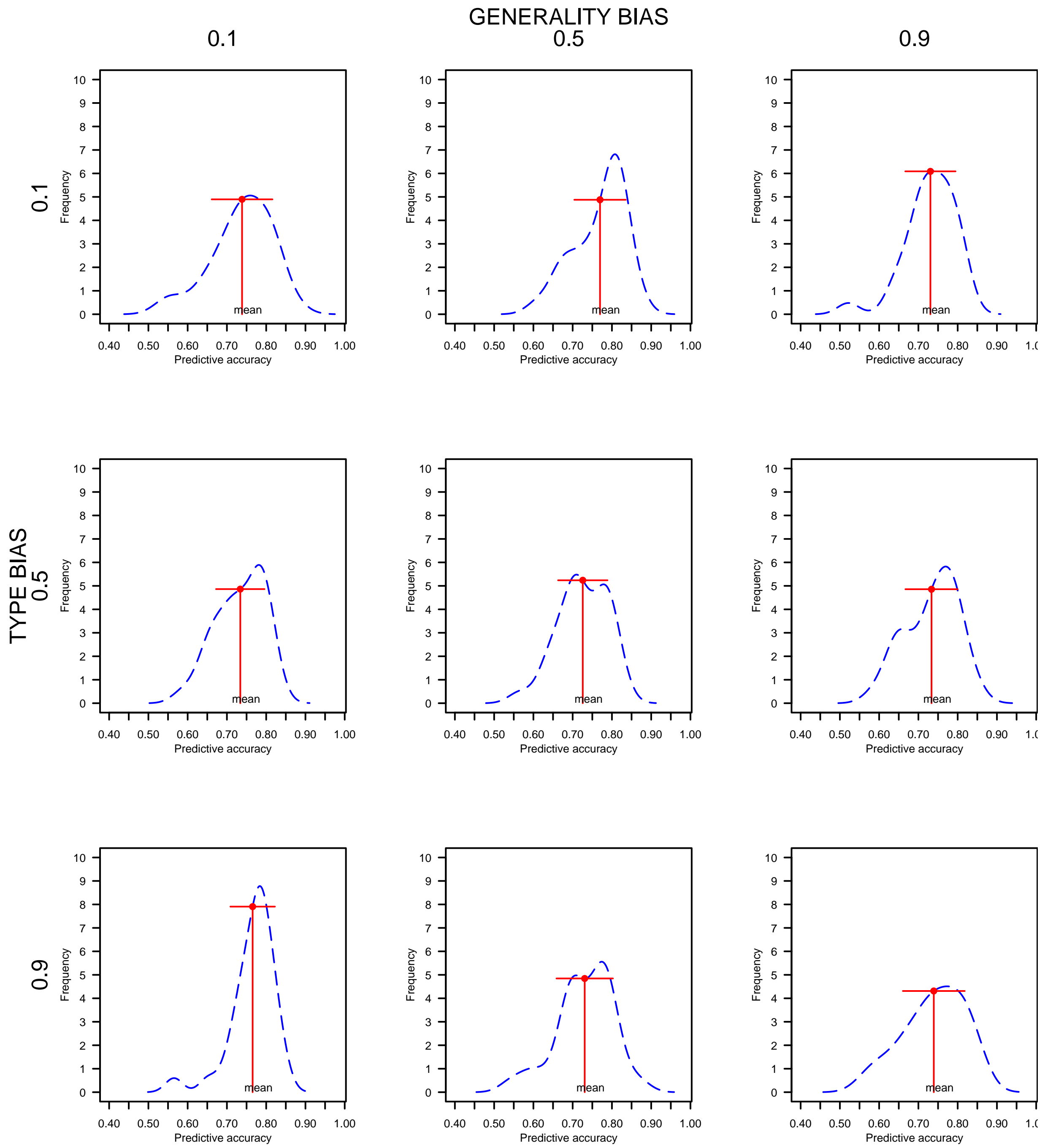


Figure A.2: Variation in generality and detector type bias (document classification)

Appendix B

Implementation details and source code

B.1 Implementation details

All the code was written on a 1.4GHz Athlon Linux box, originally in C and then in C++, compiled using g++ v2.95.3 with level 2 optimisation, and released under the GNU General Public License. The experiments described in **Chapters 4** and **5** were carried out on the Linux box already mentioned and on four 1.8GHz Pentium 4's also running Linux. Output data was processed and analysed using the R statistical package [51], with which all the graphs contained in the dissertation were also produced. Figures were created using xfig and the flowcharts with TCM. The dissertation was typeset using the LaTeX 2_ε document preparation system and the BibTeX bibliographic database system.

The code is organised into header and implementation files to aid readability, extensibility and testing. The main routines for AIS concept learner can be found in ConceptLearner.h and ConceptLearner.cpp, with specific routines for the evolutionary algorithm and classifier used by the concept learner found in EvolutionaryAlgorithm.h and Classifier.h and their associated implementation files respectively. The DataSet.h and DataSet.cpp files contain routines for handling the input, output and creation of crossvalidation sets for the voting problem, and the FeatureExtractor.h and FeatureExtractor.cpp files the feature extractor routines for the HTML document classification task. The naive Bayesian classifier is implemented in NaiveBayesianClassifier.h and NaiveBayesianClassifier.cpp. Examples of how all these routines are used can be found in crossvalidate-potter.cpp and evolve-pazzani.cpp programs, which were used to generate the results presented in **Chapters 4** and **5** respectively.

The concept learner, feature extractor, naive Bayesian classifier and data sets were instantiated as C++ classes as summarised in **Table B.1**. More details of the objects and methods contained in these classes can be found in the source code listings.

class	description	file
ConceptLearner	concept learner	ConceptLearner.h
Genome	genome	EvolutionaryAlgorithm.h
Species	species	EvolutionaryAlgorithm.h
Detector	detector	Classifier.h
FeatureVector	feature vector	DataSet.h
DataSet	voting data set	DataSet.h
CrossvalidationSet	voting crossvalidation set	DataSet.h
FeatureExtractor	feature extractor	FeatureExtractor.h
NaiveBayesianClassifier	naive Bayesian classifier	NaiveBayesianClassifier.h

Table B.1: C++ classes.

The code was tested by carefully observing the behaviour of its constituent functions on a number of artificially-generated test problems. Memory usage was assessed by observing the processes that were generated at run-time to ensure no memory leaks occured. The figures given in **Chapter 6** for run-time performance were taken from runs on the 1.4GHz Athlon box. The system itself should be viewed as an experimental version and was written so that a range of parameters could be easily set and functions interchanged. In order to provide reasonable performance levels, the code was partially optimised for speed, and therefore some functions are somewhat longer than would necessarily be found in a production version. In its current state, the system is very robust and flexible. Source code can be downloaded from the author's website at http://cogs.milieu3.net/.

B.2 Source code

B.2.1 crossvalidate-potter.cpp

```
/****************************************************************************/
/* artificial immune system concept learner v1.0                            */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                     */
/* released under the gnu general public license                            */
/****************************************************************************/

/****************************************************************************/
/* crossvalidate immune-based and naive bayesian classifiers on voting data */
/****************************************************************************/

// headers
#include <string>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "ConceptLearner.h"
#include "NaiveBayesianClassifier.h"

// constants
#define RANDSEED             100  // random seed
#define MAXFILELEN           100  // max filename length

int main(int argc, char **argv)
{
   register unsigned int i, j;
   FILE *datafile;
   time_t *start_time, *end_time;

   // save time info
   start_time = (time_t *) malloc(sizeof(time_t));
   end_time = (time_t *) malloc(sizeof(time_t));
   time(start_time);

   // parse command line
   if(argc != 8)
   {
      cerr << "crossvalidate-potter " << \
            "TRIALS CVSETS GENERATIONS INIT_BIAS TYPE_BIAS INFILE OUTFILE\n";
      return(EXIT_FAILURE);
   }
   unsigned int trials = atoi(argv[1]);
   unsigned int cvsets = atoi(argv[2]);
   unsigned int generations = atoi(argv[3]);
```

```
double initBias = atof(argv[4]);
double typeBias = atof(argv[5]);
string *infile = new string(argv[6]);
string *outfile = new string(argv[7]);

outfile->append("-");
outfile->append(argv[1]);
outfile->append("-");
outfile->append(argv[2]);
outfile->append("-");
outfile->append(argv[3]);
outfile->append("-");
outfile->append(argv[4]);
outfile->append("-");
outfile->append(argv[5]);
outfile->append(".cv.dat");

datafile = fopen(outfile->c_str(), "w");

// reset random number seed
srand(RANDSEED);
srand48(RANDSEED);

// load training set
DataSet *dataSet = new DataSet(infile);
cout << "\n*** data set loaded ***\n";

fprintf(datafile, "%-2d %-3d %-5d %-.10f %-.10f %-3d\n", \
      trials, cvsets, generations, initBias, typeBias, \
      dataSet->vectorLength);

// create crossvalidation sets
CrossvalidationSet *crossvalidationSet = new CrossvalidationSet(dataSet, \
      cvsets);
cout << "\n*** crossvalidation sets created ***\n";

// create naive bayesian classifier
NaiveBayesianClassifier *naiveBayesianClassifier = new \
      NaiveBayesianClassifier(dataSet->vectorLength);

// create concept learner
ConceptLearner *conceptLearner = new ConceptLearner();
conceptLearner->setFeatureVectorLength(dataSet->vectorLength);
conceptLearner->outputData = 1;
conceptLearner->outputStream = datafile;
conceptLearner->setGeneralityBias(initBias);
conceptLearner->setDetectorTypeBias(typeBias);
conceptLearner->save(datafile);
cout << "\n*** concept learner initialised ***\n";
```

```
cout << "\n*** performing " << trials << " trials ***\n";
for(i = 1; i <= trials; i++)
{
   crossvalidationSet->randomise();
   cout << "\n*** crossvalidating concept learner ***\n";
   for(j = 0; j < cvsets; j++)
   {
      cout << "*** trial " << i << " crossvalidation set " << \
            j << " ***\n";

      // initialise
      conceptLearner->reset();
      conceptLearner->trainingSet = crossvalidationSet->trainingSet[j];
      conceptLearner->testSet = crossvalidationSet->testSet[j];
      conceptLearner->addRandomSpecies();

      // evolve classifier
      conceptLearner->evolveClassifier(generations);

      // output data
      printf("*** final %-.10f\t%-.10f\t%-2d ***\n",
            conceptLearner->classifierTrainingFitness, \
            conceptLearner->classifierTestFitness, \
            conceptLearner->classifierSize \
            );
      fprintf(datafile,"%-.10f\t%-.10f\t%-2d\n",
            conceptLearner->classifierTrainingFitness, \
            conceptLearner->classifierTestFitness, \
            conceptLearner->classifierSize \
            );

      conceptLearner->createFromBestTraining();
      printf("*** train %-.10f\t%-.10f\t%-2d ***\n", \
            conceptLearner->testOnTrainingSet(), \
            conceptLearner->testOnTestSet(), \
            conceptLearner->bestClassifierTrainingSize \
            );
      fprintf(datafile, "%-.10f\t%-.10f\t%-2d\n", \
            conceptLearner->testOnTrainingSet(), \
            conceptLearner->testOnTestSet(), \
            conceptLearner->bestClassifierTrainingSize \
            );

      conceptLearner->createFromBestTest();
      printf("*** test %-.10f\t%-.10f\t%-2d ***\n", \
            conceptLearner->testOnTrainingSet(), \
            conceptLearner->testOnTestSet(), \
            conceptLearner->bestClassifierTestSize \
```

```
            );
        fprintf(datafile, "%-.10f\t%-.10f\t%-2d\n", \
            conceptLearner->testOnTrainingSet(), \
            conceptLearner->testOnTestSet(), \
            conceptLearner->bestClassifierTestSize \
            );

        // train naive bayesian classifier
        naiveBayesianClassifier->train(crossvalidationSet->trainingSet[j]);
        printf("*** nbc train %-.10f\ttest %-.10f ***\n\n", \
            naiveBayesianClassifier->test(crossvalidationSet-> \
            trainingSet[j]), naiveBayesianClassifier->test( \
            crossvalidationSet->testSet[j]) \
            );
        fprintf(datafile, "%-.10f\t%-.10f\n", \
            naiveBayesianClassifier->test(crossvalidationSet-> \
            trainingSet[j]), naiveBayesianClassifier-> \
            test(crossvalidationSet->testSet[j]) \
            );

        // flush data file
        fflush(datafile);
      }
   }

   fclose(datafile);

   // output time info
   time(end_time);
   printf("*** done (%d minutes) ***\a\n", \
        (unsigned int) (difftime(*end_time, *start_time) / 60.0));

   return(EXIT_SUCCESS);
}
```

B.2.2 evolve-pazzani.cpp

```
/*************************************************************************/
/* artificial immune system concept learner v1.0                        */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                  */
/* released under the gnu general public license                         */
/*************************************************************************/

/*************************************************************************/
/* evolve immune-based and naive bayesian classifiers on html data       */
/*************************************************************************/

// headers
#include <string>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "ConceptLearner.h"
#include "NaiveBayesianClassifier.h"
#include "FeatureExtractor.h"

// constants
#define RANDSEED            100  // random seed
#define MAXFILELEN          100  // max filename length

int main(int argc, char **argv)
{
   register unsigned int i;
   FILE *datafile;
   time_t *start_time, *end_time;
   DataSet *trainingSet, *testSet;

   // save time info
   start_time = (time_t *) malloc(sizeof(time_t));
   end_time = (time_t *) malloc(sizeof(time_t));
   time(start_time);

   // parse command line
   if(argc != 9)
   {
      cerr << "evolve-pazzani " << \
            "TRIALS TRAINING-SIZE GENERATIONS INIT_BIAS TYPE_BIAS " << \
            "FV_LENGTH INFILE OUTFILE\n";
      return(EXIT_FAILURE);
   }
   unsigned int trials = atoi(argv[1]);
   unsigned int trainingSize = atoi(argv[2]);
```

```
unsigned int generations = atoi(argv[3]);
double initBias = atof(argv[4]);
double typeBias = atof(argv[5]);
unsigned int fvLength = atoi(argv[6]);
string *infile = new string(argv[7]);
string *outfile = new string(argv[8]);

outfile->append("-");
outfile->append(argv[1]);
outfile->append("-");
outfile->append(argv[2]);
outfile->append("-");
outfile->append(argv[3]);
outfile->append("-");
outfile->append(argv[4]);
outfile->append("-");
outfile->append(argv[5]);
outfile->append("-");
outfile->append(argv[6]);
outfile->append(".ev.dat");

datafile = fopen(outfile->c_str(), "w");

// reset random number seed
srand(RANDSEED);
srand48(RANDSEED);

// load data set
FeatureExtractor *featureExtractor = new FeatureExtractor(infile);
cout << "\n*** html data loaded and parsed ***\n";
featureExtractor->vectorLength = fvLength;

fprintf(datafile, "%-2d %-3d %-5d %-.10f %-.10f %-3d\n", \
      trials, trainingSize, generations, initBias, typeBias, fvLength);

// create naive bayesian classifier
NaiveBayesianClassifier *naiveBayesianClassifier = new \
      NaiveBayesianClassifier(fvLength);

// create concept learner
ConceptLearner *conceptLearner = new ConceptLearner();
conceptLearner->setFeatureVectorLength(fvLength);
conceptLearner->outputData = 1;
conceptLearner->outputStream = datafile;
conceptLearner->setGeneralityBias(initBias);
conceptLearner->setDetectorTypeBias(typeBias);
conceptLearner->save(datafile);
cout << "\n*** concept learner initialised ***\n";
```

```
cout << "\n*** performing " << trials << " trials ***\n";
for(i = 1; i <= trials; i++)
{
   featureExtractor->createDataSet(fvLength, trainingSize);
   trainingSet = featureExtractor->trainingSet;
   testSet = featureExtractor->testSet;

   cout << "*** trial " << i << " ***\n";

   // initialise
   conceptLearner->reset();
   conceptLearner->trainingSet = trainingSet;
   conceptLearner->testSet = testSet;
   conceptLearner->addRandomSpecies();

   // evolve classifier
   conceptLearner->evolveClassifier(generations);

   // output data
   printf("*** final %-.10f\t%-.10f\t%-2d ***\n",
         conceptLearner->classifierTrainingFitness, \
         conceptLearner->classifierTestFitness, \
         conceptLearner->classifierSize \
         );
   fprintf(datafile,"%-.10f\t%-.10f\t%-2d\n",
         conceptLearner->classifierTrainingFitness, \
         conceptLearner->classifierTestFitness, \
         conceptLearner->classifierSize \
         );

   conceptLearner->createFromBestTraining();
   printf("*** train %-.10f\t%-.10f\t%-2d ***\n", \
         conceptLearner->testOnTrainingSet(), \
         conceptLearner->testOnTestSet(), \
         conceptLearner->bestClassifierTrainingSize \
         );
   fprintf(datafile, "%-.10f\t%-.10f\t%-2d\n", \
         conceptLearner->testOnTrainingSet(), \
         conceptLearner->testOnTestSet(), \
         conceptLearner->bestClassifierTrainingSize \
         );

   conceptLearner->createFromBestTest();
   printf("*** test %-.10f\t%-.10f\t%-2d ***\n", \
         conceptLearner->testOnTrainingSet(), \
         conceptLearner->testOnTestSet(), \
         conceptLearner->bestClassifierTestSize \
         );
   fprintf(datafile, "%-.10f\t%-.10f\t%-2d\n", \
```

```
            conceptLearner->testOnTrainingSet(), \
            conceptLearner->testOnTestSet(), \
            conceptLearner->bestClassifierTestSize \
            );

        // train naive bayesian classifier
        naiveBayesianClassifier->train(trainingSet);
        printf("*** nbc train %-.10f\ttest %-.10f ***\n\n", \
            naiveBayesianClassifier->test(trainingSet), \
            naiveBayesianClassifier->test(testSet));
        fprintf(datafile, "%-.10f\t%-.10f\n", \
            naiveBayesianClassifier->test(trainingSet), \
            naiveBayesianClassifier->test(testSet));

        // flush data file
        fflush(datafile);
    }

    fclose(datafile);

    // output time info
    time(end_time);
    printf("*** done (%d minutes) ***\a\n", \
        (unsigned int) (difftime(*end_time, *start_time) / 60.0));

    return(EXIT_SUCCESS);
}
```

B.2.3 Classifier.h

```
/*************************************************************************/
/* artificial immune system concept learner v1.0                          */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                   */
/* released under the gnu general public license                          */
/*************************************************************************/

/*************************************************************************/
/* classifier routines                                                    */
/*************************************************************************/

#ifndef CLASSIFIER_H
#define CLASSIFIER_H

// headers
#include <iostream>
#include <stdio.h>

// constants
#define SELF                0  // self class
#define NONSELF             1  // nonself class
#define MASKVALUE           2  // mask value

//
// detector class
//
class Detector
{
   public:
      Detector(const unsigned int length);
      Detector::~Detector(void);

      unsigned int length;  // length of detector vector
      unsigned int *value;  // vector values
      double threshold;  // detector threshold
      unsigned int type;  // detector type

      void save(FILE *outputStream);  // save to stream
      void show(void) { save(stdout); };  // output to stdout
};

#endif
```

B.2.4 Classifier.cpp

```
/*****************************************************************************/
/* artificial immune system concept learner v1.0                            */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                     */
/* released under the gnu general public license                            */
/*****************************************************************************/

/*****************************************************************************/
/* classifier routines                                                       */
/*****************************************************************************/

// headers
#include "Classifier.h"

//
// detector class public methods
//

// constructor - initialise detector
// length - length of detector
Detector::Detector(const unsigned int length)
{
   this->length = length;
   threshold = 0.0;
   value = new unsigned int [length];
   type = 0;
}

// destructor
Detector::~Detector(void)
{
   delete [] value;
}

// save to stream
// outputStream - stream to save to
void Detector::save(FILE *outputStream)
{
   register unsigned int i;

   fprintf(outputStream, \
      "%-3d %-.10f %-1d\n", \
      length, \
      threshold, \
      type \
      );

   for(i = 0; i < length; i++)
```

```
        fprintf(outputStream, "%-1d ", value[i]);
    fprintf(outputStream, "\n");

    fflush(outputStream);
}
```

B.2.5 ConceptLearner.h

```
/*****************************************************************************/
/* artificial immune system concept learner v1.0                           */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                     */
/* released under the gnu general public license                           */
/*****************************************************************************/

/*****************************************************************************/
/* concept learner routines                                                 */
/*****************************************************************************/

#ifndef CONCEPTLEARNER_H
#define CONCEPTLEARNER_H

// headers
#include "EvolutionaryAlgorithm.h"
#include "Classifier.h"
#include "DataSet.h"
#include <stdio.h>

// constants
#define DEF_SPECIES_SIZE                  100  // default species size
#define DEF_MAX_FEATURE_VECTOR_LENGTH     256  // default max feature vector len
#define DEF_MAX_CLASSIFIER_SIZE           20  // default max classifier size

//
// concept learner class
//
class ConceptLearner
{
   public:
      ConceptLearner(const unsigned int speciesSize = DEF_SPECIES_SIZE, \
            const unsigned int maxFeatureVectorLength = \
            DEF_MAX_FEATURE_VECTOR_LENGTH, \
            const unsigned int maxClassifierSize = DEF_MAX_CLASSIFIER_SIZE);
      ~ConceptLearner(void);

      unsigned int maxClassifierSize;
      unsigned int maxFeatureVectorLength;

      double generalityBias;
      double detectorTypeBias;
      short unsigned int outputData;  // output data to file (flag; def = 0)
      FILE *outputStream;  // stream to output data to

      unsigned int featureVectorLength;
      unsigned int speciesSize;
```

```
    double removalThreshold;
    unsigned int removalCount, removalGenerations;
    double creationThreshold;
    unsigned int creationCount, creationGenerations;

    double lastClassifierTrainingFitness;

    unsigned int numSpecies;  // current number of species
    unsigned int classifierSize;  // current classifier size
    unsigned int bestClassifierTrainingSize, bestClassifierTestSize;

    double classifierTrainingFitness, classifierTestFitness;
    double bestClassifierTrainingFitness, bestClassifierTestFitness;
    unsigned int falsePositives, falseNegatives;

    Species **parent;  // parent population
    Species **child;  // child population
    DataSet *trainingSet;
    DataSet *testSet;

    void evolveClassifier(const unsigned int generations = 1);
    double testOnTrainingSet();
    double testOnTestSet();
    void save(FILE *outputStream);
    void saveEvolutionData(FILE *outputStream);
    void saveEvolutionData(FILE *outputStream, const unsigned int index);
    void addRandomSpecies(void);
    void reset(void);
    void createFromBestTraining(void);
    void createFromBestTest(void);
    void setGeneralityBias(const double generalityBias);
    void setDetectorTypeBias(const double detectorTypeBias);
    void setFeatureVectorLength(const unsigned int featureVectorLength);
    void saveClassifier(FILE *outputStream);
    void restoreEvolvedClassifier(void);
    void show(void) { save(stdout); };
    void showEvolutionData(void) { saveEvolutionData(stdout); };
    void showEvolutionData(const unsigned int index)
          { saveEvolutionData(stdout, index); };

private:
    Species **species1, **species2;
    Detector **detector;
    unsigned int *detectorActivity;
    Genome **bestTrainingGenome;
    Genome **bestTestGenome;

    void evaluateCCA(void);
    void evaluateSpecies(const unsigned int index);
```

```
    void breedCCA(void);
    void breedSpecies(const unsigned int index);
    void handleCCAStagnation(void);
    void removeStagnatedSpecies(void);
    unsigned int classify(FeatureVector *featureVector);
    void swapPopulations(void)
        { Species **tmp = parent; parent = child; child = tmp; };
};

#endif
```

B.2.6 ConceptLearner.cpp

```
/*************************************************************************/
/* artificial immune system concept learner v1.0                         */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                  */
/* released under the gnu general public license                         */
/*************************************************************************/

/*************************************************************************/
/* concept learner routines                                              */
/*************************************************************************/

// headers
#include "ConceptLearner.h"
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

//
// concept learner class public methods
//

// constructor - create concept learner
// speciesSize - max num of species
ConceptLearner::ConceptLearner(const unsigned int speciesSize, \
      const unsigned int maxFeatureVectorLength, \
      const unsigned int maxClassifierSize)
{
   register unsigned int i = maxClassifierSize;

   this->speciesSize = speciesSize;
   this->maxFeatureVectorLength = featureVectorLength = maxFeatureVectorLength;
   this->maxClassifierSize = maxClassifierSize;

   // set defaults
   classifierSize = numSpecies = 0;
   lastClassifierTrainingFitness = 0.0;
   classifierTrainingFitness = classifierTestFitness = 0.0;
   bestClassifierTrainingFitness = bestClassifierTestFitness = 0.0;
   bestClassifierTrainingSize = bestClassifierTestSize = 0;

   outputData = 0;

   removalThreshold = creationThreshold = 0.001;
   removalGenerations = creationGenerations = removalCount = creationCount = 2;
```

```
    generalityBias = 0.5;
    detectorTypeBias = 0.5;

    // allocate memory
    parent = new Species * [maxClassifierSize];
    child = new Species * [maxClassifierSize];
    detector = new Detector * [maxClassifierSize];
    detectorActivity = new unsigned int [maxClassifierSize];
    bestTrainingGenome = new Genome * [maxClassifierSize];
    bestTestGenome = new Genome * [maxClassifierSize];
    while(i--)
    {
        parent[i] = new Species(speciesSize, maxFeatureVectorLength);
        child[i] = new Species(speciesSize, maxFeatureVectorLength);
        detector[i] = new Detector(maxFeatureVectorLength);
        bestTrainingGenome[i] = new Genome(maxFeatureVectorLength);
        bestTestGenome[i] = new Genome(maxFeatureVectorLength);
    }
}

// destructor
ConceptLearner::~ConceptLearner(void)
{
    register unsigned int i = maxClassifierSize;

    while(i--)
    {
        delete parent[i];
        delete child[i];
        delete detector[i];
        delete bestTrainingGenome[i];
        delete bestTestGenome[i];
    }
    delete [] bestTestGenome;
    delete [] bestTrainingGenome;
    delete [] detectorActivity;
    delete [] detector;
    delete [] child;
    delete [] parent;
}

// evolve concept learner
// generations - generation to evolve for
void ConceptLearner::evolveClassifier(const unsigned int generations)
{
    register unsigned int i = 0;
    register unsigned int ngens = generations;
    register short unsigned int flag = outputData;
```

```
    // output some info
    printf("%-4d\t%-.10f\t%-.10f\t%-2d\t%-.10f\t%-2d\t%-.10f\t%-2d\n", \
           i, classifierTrainingFitness, classifierTestFitness, \
           classifierSize, \
           bestClassifierTrainingFitness, bestClassifierTrainingSize, \
           bestClassifierTestFitness, bestClassifierTestSize);


    // output data if required
    if(flag)
       saveEvolutionData(outputStream, 0);

    // evolve cca
    for(i = 1; i <= ngens; i++)
    {

       // output some info
       printf("%-4d\t%-.10f\t%-.10f\t%-2d\t%-.10f\t%-2d\t%-.10f\t%-2d\n", \
              i, classifierTrainingFitness, classifierTestFitness, \
              classifierSize, \
              bestClassifierTrainingFitness, bestClassifierTrainingSize, \
              bestClassifierTestFitness, bestClassifierTestSize);


       // breed cca
       breedCCA();

       // handle cca stagnation
       handleCCAStagnation();

       // output data if required
       if(flag)
          saveEvolutionData(outputStream, i);
    }
}

// calculate predictive accuracy of concept learner on training set
double ConceptLearner::testOnTrainingSet()
{
    register unsigned int i = classifierSize, tmp, fpos, fneg;

    // reset detector activities
    while(i--)
       detectorActivity[i] = 0;

    // classify each training example
    fpos = fneg = 0;
    i = trainingSet->size;
    while(i--)
```

```
        if((tmp = classify(trainingSet->featureVector[i])) != trainingSet->\
                vectorClass[i])
        {
            if(tmp == 0)
                fneg++;
            else
                fpos++;
        }

    return(double(trainingSet->size - fpos - fneg) / double(trainingSet->size));
}

// calculate predictive accuracy of concept learner on test set
double ConceptLearner::testOnTestSet()
{
    register unsigned int i = classifierSize, tmp, fpos, fneg;

    // reset detector activities
    while(i--)
        detectorActivity[i] = 0;

    // classify each test example
    fpos = fneg = 0;
    i = testSet->size;
    while(i--)
        if((tmp = classify(testSet->featureVector[i])) != testSet->\
                vectorClass[i])
        {
            if(tmp == 0)
                fneg++;
            else
                fpos++;
        }

    return(double(testSet->size - fpos - fneg) / double(testSet->size));
}

// save concept learner to stream
void ConceptLearner::save(FILE *outputStream)
{
    fprintf(outputStream, \
            "%-3d %-3d %-3d %-3d %-.5f %-3d %-3d %-.5f %-2d %-2d %-.5f %-2d %-2d\n", \
            numSpecies, \
            classifierSize, \
            maxClassifierSize, \
            maxFeatureVectorLength, \
            generalityBias, \
            featureVectorLength, \
            speciesSize, \
```

```
            removalThreshold, \
            removalGenerations, \
            removalCount, \
            creationThreshold, \
            creationGenerations, \
            creationCount
            );
}

// save evolution data to stream
void ConceptLearner::saveEvolutionData(FILE *outputStream)
{
    register unsigned int i;

    fprintf(outputStream, \
            "%-.10f %-.10f %-2d %-.10f %-2d %-.10f %-2d %-.10f\n", \
            classifierTrainingFitness,
            classifierTestFitness, \
            classifierSize, \
            bestClassifierTrainingFitness, \
            bestClassifierTrainingSize, \
            bestClassifierTestFitness, \
            bestClassifierTestSize, \
            lastClassifierTrainingFitness \
            );

    for(i = 0; i < numSpecies; i++)
    {
        fprintf(outputStream, \
                "%-4d\t%-3d\t%-.10f\t%-10.10f\n", \
                i, \
                parent[i]->fittestIndividual, \
                parent[i]->genome[parent[i]->fittestIndividual]->fitness, \
                parent[i]->meanSpeciesFitness \
                );
    }

    fflush(outputStream);
}

// save evolution data with index to stream
void ConceptLearner::saveEvolutionData(FILE *outputStream, \
        const unsigned int index)
{
    fprintf(outputStream, "%-4d ", index);
    saveEvolutionData(outputStream);
}

// add a new random species
```

```
void ConceptLearner::addRandomSpecies(void)
{
   register unsigned int i;

   // if less than maximum species size
   if(numSpecies < maxClassifierSize)
   {
      // add new parent species
      parent[numSpecies]->randomise();
      numSpecies++;
      evaluateSpecies(numSpecies - 1);

      // update statistics
      i = numSpecies;
      while(i--)
         parent[i]->genome[parent[i]->fittestIndividual]-> \
               setDetector(detector[i]);
      classifierSize = numSpecies;

      // calculate fitness on training and test sets
      classifierTestFitness = testOnTestSet();
      classifierTrainingFitness = testOnTrainingSet();

      // save fitnesses and genomes if best
      if((classifierTrainingFitness > bestClassifierTrainingFitness) || \
            ((classifierTrainingFitness == bestClassifierTrainingFitness) && \
            (classifierSize < bestClassifierTrainingSize)))
      {
         // save as best fitness on training set
         bestClassifierTrainingFitness = classifierTrainingFitness;
         bestClassifierTrainingSize = classifierSize;
         i = classifierSize;
         while(i--)
         {
            bestTrainingGenome[i]->copyGenome(parent[i]-> \
                  genome[parent[i]->fittestIndividual]);
         }
      }
      if((classifierTestFitness > bestClassifierTestFitness) || \
            ((classifierTestFitness == bestClassifierTestFitness) && \
            (classifierSize < bestClassifierTestSize)))
      {
         // save as best fitness on test set
         bestClassifierTestFitness = classifierTestFitness;
         bestClassifierTestSize = classifierSize;
         i = classifierSize;
         while(i--)
         {
            bestTestGenome[i]->copyGenome(parent[i]-> \
```

```
                  genome[parent[i]->fittestIndividual]);
            }
        }
    }
}

// reset concept learner
void ConceptLearner::reset(void)
{
    classifierSize = numSpecies = 0;
    lastClassifierTrainingFitness = 0.0;
    classifierTrainingFitness = classifierTestFitness = 0.0;
    bestClassifierTrainingFitness = bestClassifierTestFitness = 0.0;
    bestClassifierTrainingSize = bestClassifierTestSize = 0;
    removalCount = removalGenerations;
    creationCount = creationGenerations;
}

// create a classifier from best solution on training set so far
void ConceptLearner::createFromBestTraining(void)
{
    register unsigned int i = bestClassifierTrainingSize;

    while(i--)
        bestTrainingGenome[i]->setDetector(detector[i]);
    classifierSize = bestClassifierTrainingSize;
}

// create a classifier from best solution on test set so far
void ConceptLearner::createFromBestTest(void)
{
    register unsigned int i = bestClassifierTestSize;

    while(i--)
        bestTestGenome[i]->setDetector(detector[i]);
    classifierSize = bestClassifierTestSize;
}

// set generality bias
void ConceptLearner::setGeneralityBias(const double generalityBias)
{
    register unsigned int i = maxClassifierSize, j;

    // set for all genomes
    this->generalityBias = generalityBias;
    while(i--)
    {
        j = speciesSize;
        while(j--)
```

```
        {
            parent[i]->genome[j]->generalityBias = \
                    child[i]->genome[j]->generalityBias = generalityBias;
        }
        bestTrainingGenome[i]->generalityBias = generalityBias;
        bestTestGenome[i]->generalityBias = generalityBias;
    }

}

// set detector type bias
void ConceptLearner::setDetectorTypeBias(const double detectorTypeBias)
{
    register unsigned int i = maxClassifierSize, j;

    // set for all genomes
    this->detectorTypeBias = detectorTypeBias;
    while(i--)
    {
        j = speciesSize;
        while(j--)
        {
            parent[i]->genome[j]->typeBias = \
                    child[i]->genome[j]->typeBias = detectorTypeBias;
        }
        bestTrainingGenome[i]->typeBias = detectorTypeBias;
        bestTestGenome[i]->typeBias = detectorTypeBias;
    }

}

// set feature vector length
void ConceptLearner::setFeatureVectorLength( \
        const unsigned int featureVectorLength)
{
    register unsigned int i = maxClassifierSize, j;

    this->featureVectorLength = featureVectorLength;
    // set for all genomes
    while(i--)
    {
        j = speciesSize;
        while(j--)
        {
            parent[i]->genome[j]->patternLength = \
                    child[i]->genome[j]->patternLength = featureVectorLength;
            parent[i]->genome[j]->size = 2 * featureVectorLength + \
                    child[i]->genome[j]->thresholdLength;
            child[i]->genome[j]->size = 2 * featureVectorLength + \
```

```
                child[i]->genome[j]->thresholdLength;
        }
        bestTrainingGenome[i]->patternLength = featureVectorLength;
        bestTestGenome[i]->patternLength = featureVectorLength;
        bestTrainingGenome[i]->size = 2 * featureVectorLength + \
                bestTrainingGenome[i]->thresholdLength;
        bestTestGenome[i]->size = 2 * featureVectorLength + \
                bestTestGenome[i]->thresholdLength;
    }

}

// save classifier to stream
void ConceptLearner::saveClassifier(FILE *outputStream)
{
    register unsigned int i;

    for(i = 0; i < numSpecies; i++)
        parent[i]->genome[parent[i]->fittestIndividual]->save(outputStream);
    for(i = 0; i < bestClassifierTrainingSize; i++)
        bestTrainingGenome[i]->save(outputStream);
    for(i = 0; i < bestClassifierTestSize; i++)
        bestTestGenome[i]->save(outputStream);
}

// create classifier from last evolved classifier
void ConceptLearner::restoreEvolvedClassifier(void)
{
    register unsigned int i = numSpecies;

    while(i--)
        parent[i]->genome[parent[i]->fittestIndividual]->setDetector(detector[i]);
    classifierSize = numSpecies;
}

//
// concept learner class private methods
//

// evaluate concept learner
void ConceptLearner::evaluateCCA(void)
{
    register unsigned int i, oldFittestIndividual;

    // evaluate species
    i = numSpecies;
    while(i--)
    {
        // save new fittest as fitness update synchronous
```

```
      oldFittestIndividual = parent[i]->fittestIndividual;
      evaluateSpecies(i);
      child[i]->fittestIndividual = parent[i]->fittestIndividual;
      parent[i]->fittestIndividual = oldFittestIndividual;
   }

   i = numSpecies;
   while(i--)
   {
      // copy new fittest and create classifier of best individuals
      parent[i]->fittestIndividual = child[i]->fittestIndividual;
      parent[i]->genome[parent[i]->fittestIndividual]->setDetector(detector[i]);
   }
   classifierSize = numSpecies;

   // calculate fitness on training and test sets
   classifierTestFitness = testOnTestSet();
   classifierTrainingFitness = testOnTrainingSet();

   // save fitnesses and genomes if best
   if((classifierTrainingFitness > bestClassifierTrainingFitness) || \
         ((classifierTrainingFitness == bestClassifierTrainingFitness) && \
         (classifierSize < bestClassifierTrainingSize)))
   {
      // save as best fitness on training set
      bestClassifierTrainingFitness = classifierTrainingFitness;
      bestClassifierTrainingSize = classifierSize;
      i = classifierSize;
      while(i--)
      {
         bestTrainingGenome[i]->copyGenome(parent[i]-> \
               genome[parent[i]->fittestIndividual]);
      }
   }
   if((classifierTestFitness > bestClassifierTestFitness) || \
         ((classifierTestFitness == bestClassifierTestFitness) && \
         (classifierSize < bestClassifierTestSize)))
   {
      // save as best fitness on test set
      bestClassifierTestFitness = classifierTestFitness;
      bestClassifierTestSize = classifierSize;
      i = classifierSize;
      while(i--)
      {
         bestTestGenome[i]->copyGenome(parent[i]-> \
               genome[parent[i]->fittestIndividual]);
      }
   }
}
```

```
// evaluate a species
// index - index of species to evaluate
void ConceptLearner::evaluateSpecies(const unsigned int index)
{
   register unsigned int i, newFittestIndividual;
   register double bestIndividualFitness, worstIndividualFitness;
   register double meanSpeciesFitness, speciesScaledFitnessSum;
   register double individualFitness;

   // calculate fitness of each individual
   newFittestIndividual = parent[index]->fittestIndividual = 0;
   i = numSpecies;
   while(i--)
      parent[i]->genome[parent[i]->fittestIndividual]->setDetector(detector[i]);
   classifierSize = numSpecies;
   parent[index]->genome[0]->fitness = \
         meanSpeciesFitness = \
         bestIndividualFitness = worstIndividualFitness = \
         testOnTrainingSet();
   i = speciesSize;
   while(--i)
   {
      parent[index]->genome[i]->setDetector(detector[index]);
      parent[index]->genome[i]->fitness = individualFitness = \
            testOnTrainingSet();
      if(individualFitness >= bestIndividualFitness)
      {
         newFittestIndividual = i;
         bestIndividualFitness = individualFitness;
      }
      else
         if(individualFitness < worstIndividualFitness)
            worstIndividualFitness = individualFitness;
      meanSpeciesFitness += individualFitness;
   }
   parent[index]->fittestIndividual = newFittestIndividual;

   // scale fitnesses
   parent[index]->meanSpeciesFitness = meanSpeciesFitness = \
         meanSpeciesFitness / double(speciesSize);
   speciesScaledFitnessSum = 0.0;
   i = speciesSize;
   while(i--)
   {
      // fitness proportionate scaling with linear balancing
      if(parent[index]->genome[i]->fitness == worstIndividualFitness)
      {
         parent[index]->genome[i]->scaledFitness = 0.0;
```

```
            continue;
        }
        if(parent[index]->genome[i]->fitness == bestIndividualFitness)
        {
            parent[index]->genome[i]->scaledFitness = 2.0;
            speciesScaledFitnessSum += 2.0;
            continue;
        }
        if(parent[index]->genome[i]->fitness == meanSpeciesFitness)
        {
            parent[index]->genome[i]->scaledFitness = 1.0;
            speciesScaledFitnessSum += 1.0;
            continue;
        }
        if(parent[index]->genome[i]->fitness > meanSpeciesFitness)
        {
            parent[index]->genome[i]->scaledFitness = 1.0 + ( \
                    (parent[index]->genome[i]->fitness - \
                    meanSpeciesFitness) / \
                    (bestIndividualFitness - \
                    meanSpeciesFitness) \
                    );
            speciesScaledFitnessSum += parent[index]->genome[i]->scaledFitness;
            continue;
        }
        parent[index]->genome[i]->scaledFitness = 1.0 - ( \
                (meanSpeciesFitness - \
                parent[index]->genome[i]->fitness) / \
                (meanSpeciesFitness - \
                worstIndividualFitness) \
                );
        speciesScaledFitnessSum += parent[index]->genome[i]->scaledFitness;
    }
    parent[index]->speciesScaledFitnessSum = speciesScaledFitnessSum;
}

// breed current parent population
void ConceptLearner::breedCCA(void)
{
    register unsigned int i = numSpecies;

    // save current training fitness
    lastClassifierTrainingFitness = classifierTrainingFitness;

    // elitism - keep best individual
    while(i--)
    {
        child[i]->genome[0]->copyGenome(parent[i]-> \
            genome[parent[i]->fittestIndividual]);
```

```
      child[i]->fittestIndividual = 0;
   }

   i = numSpecies;
   while(i--)
      breedSpecies(i);

   // make child population new parent population and evaluate
   swapPopulations();
   evaluateCCA();
}

// breed a species
// index - index of species to breed
void ConceptLearner::breedSpecies(const unsigned int index)
{
   register unsigned int i = speciesSize;
   register Genome *parent1, *parent2;

   // create children
   while(--i)
   {
      // selection
      parent1 = parent[index]->FPSelection();
      parent2 = parent[index]->FPSelection();
      // crossover
      child[index]->genome[i]->uniformCrossover(parent1, parent2);
      // mutation
      child[index]->genome[i]->mutateBinary();
   }
}

// handle creation and deletion of stagnated species
void ConceptLearner::handleCCAStagnation(void)
{
   register double fitnessStep;

   // calculate difference in fitnesses
   fitnessStep = fabs(classifierTrainingFitness - \
         lastClassifierTrainingFitness);

   // handle removal of stagnated species
   if(fitnessStep < removalThreshold)
   {
      if(removalCount)
         removalCount--;
      else
      {
         // stagnated
```

```
            removeStagnatedSpecies();
            removalCount = removalGenerations;
        }
    }
    else
        // not stagnated
        removalCount = removalGenerations;

    // handle creation of new species
    if(fitnessStep < creationThreshold)
    {
        if(creationCount)
            creationCount--;
        else
        {
            // stagnated
            addRandomSpecies();
            creationCount = creationGenerations;
        }
    }
    else
        // not stagnated
        creationCount = creationGenerations;
}

// remove stagnated species
void ConceptLearner::removeStagnatedSpecies(void)
{
    register unsigned int i = numSpecies, j;
    register short unsigned int flag = 0;

    // for each species
    while(i--)
    {
        // stop if only one species left
        if(numSpecies == 1)
            break;

        // remove if detector not active
        if(!detectorActivity[i])
        {
            for(j = i; j < classifierSize; ++j)
                parent[j]->copySpecies(parent[j + 1]);
            numSpecies--;
            flag = 1;
        }
    }
    // if species removed
    if(flag)
```

```
    {
        // update statistics
        i = numSpecies;
        while(i--)
            parent[i]->genome[parent[i]->fittestIndividual]-> \
                    setDetector(detector[i]);
        classifierSize = numSpecies;

        // calculate fitness on training and test sets
        classifierTestFitness = testOnTestSet();
        classifierTrainingFitness = testOnTrainingSet();

        // save fitnesses and genomes if best
        if((classifierTrainingFitness > bestClassifierTrainingFitness) || \
                ((classifierTrainingFitness == bestClassifierTrainingFitness) && \
                (classifierSize < bestClassifierTrainingSize)))
        {
            // save as best fitness on training set
            bestClassifierTrainingFitness = classifierTrainingFitness;
            bestClassifierTrainingSize = classifierSize;
            i = classifierSize;
            while(i--)
            {
                bestTrainingGenome[i]->copyGenome(parent[i]-> \
                        genome[parent[i]->fittestIndividual]);
            }
        }
        if((classifierTestFitness > bestClassifierTestFitness) || \
                ((classifierTestFitness == bestClassifierTestFitness) && \
                (classifierSize < bestClassifierTestSize)))
        {
            // save as best fitness on test set
            bestClassifierTestFitness = classifierTestFitness;
            bestClassifierTestSize = classifierSize;
            i = classifierSize;
            while(i--)
            {
                bestTestGenome[i]->copyGenome(parent[i]-> \
                        genome[parent[i]->fittestIndividual]);
            }
        }
    }
}

// classify a feature vector
unsigned int ConceptLearner::classify(FeatureVector *featureVector)
{
    register unsigned int i = classifierSize, j, maskBits, matchingBits;
    register double affinity;
```

```
    // for each detector
    while(i--)
    {
        // count num of matching non-mask bits
        maskBits = matchingBits = 0;
        j = detector[i]->length;
        while(j--)
        {
            if(detector[i]->value[j] == MASKVALUE)
                maskBits++;
            else
                if(detector[i]->value[j] == featureVector->value[j])
                    matchingBits++;
        }
        // calculate affinity
        if(detector[i]->length == maskBits)
            affinity = 1.0;
        else
            affinity = double(matchingBits) / double(detector[i]->length \
                - maskBits);
        if(affinity > detector[i]->threshold)
        {
            // a match
            detectorActivity[i]++;
            if(detector[i]->type == SELF)
                return(SELF);
            else
                return(NONSELF);
        }
    }
    // no match
    return(SELF);
}
```

B.2.7 DataSet.h

```
/******************************************************************************/
/* artificial immune system concept learner v1.0                             */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                      */
/* released under the gnu general public license                             */
/******************************************************************************/

/******************************************************************************/
/* data set routines                                                          */
/******************************************************************************/

#ifndef DATASET_H
#define DATASET_H

// headers
#include <string>
#include <stdio.h>

//
// feature vector class
//
class FeatureVector
{
   public:
      FeatureVector(const unsigned int length);
      ~FeatureVector(void);

      unsigned int length;  // vector length
      unsigned int *value;  // vector values

      void save(FILE *outputStream);  // save to stream
      void show(void) { save(stdout); };  // output to stdout
};

//
// data set class
//
class DataSet
{
   public:
      DataSet(const unsigned int size, const unsigned int vectorLength);
      DataSet(string *infile);
      ~DataSet(void);

      unsigned int size;  // num of samples
      unsigned int vectorLength;  // sample vector length
      FeatureVector **featureVector;  // sample vectors
      string **filename;  // filenames of input documents
      unsigned int *vectorClass;  // vector classes
```

```
    void save(FILE *outputStream);
    void show(void) { save(stdout); };
    void copySample(DataSet *dataSet, const unsigned int fromIndex, \
         const unsigned int toIndex);
    void swapSample(DataSet *dataSet, const unsigned int fromIndex, \
         const unsigned int toIndex);
    void split(DataSet *dataSet1, DataSet *dataSet2, \
         const unsigned int dataSet1Size);
    void randomise(void);
};

//
// crossvalidation set class
//
class CrossvalidationSet
{
  public:
    CrossvalidationSet(DataSet *dataSet, const unsigned int cvsets);

    unsigned int cvsets;  // num of crossvalidation sets
    DataSet **trainingSet, **testSet;

    void randomise(void);
};

#endif
```

B.2.8 DataSet.cpp

```
/****************************************************************************/
/* artificial immune system concept learner v1.0                           */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                    */
/* released under the gnu general public license                           */
/****************************************************************************/

/****************************************************************************/
/* data set routines                                                       */
/****************************************************************************/

// headers
#include "DataSet.h"
#include <fstream>
#include <iostream>
#include "Classifier.h"

// constants
#define MAXLINELEN      1000  // max input line length

//
// feature vector class public methods
//

// constructor - create feature vector
// length - length of vector
FeatureVector::FeatureVector(const unsigned int length)
{
   value = new unsigned int [length];
   this->length = length;
}

// destructor
FeatureVector::~FeatureVector(void)
{
   delete [] value;
}

// save to stream
void FeatureVector::save(FILE *outputStream)
{
   register unsigned int i;

   fprintf(outputStream, "%-4d", length);
   for(i = 0; i < length; i++)
      fprintf(outputStream, " %-2d", value[i]);
   fprintf(outputStream, "\n");
```

```
    fflush(outputStream);
}

//
// data set class public methods
//

// constructor - create data set
// size - num of vectors in data set
// vectorLength - len of feature vectors
DataSet::DataSet(const unsigned int size, const unsigned int vectorLength)
{
    register unsigned int i;

    this->size = size;
    this->vectorLength = vectorLength;
    featureVector = new FeatureVector * [size];
    vectorClass = new unsigned int [size];
    i = size;
    while(i--)
        featureVector[i] = new FeatureVector(vectorLength);
}

// constructor - create dataset from input file
DataSet::DataSet(string *infile)
{
    register unsigned int i, j, lines = 0, featureVectorLength;
    char line[MAXLINELEN];
    ifstream *instream = new ifstream();;

    instream->open(infile->c_str());

    // count num of lines
    instream->getline(line, MAXLINELEN);
    featureVectorLength = vectorLength = strlen(line) - 1;
    while(instream->getline(line, MAXLINELEN))
        lines++;
    instream->close();

    // initialise memory
    featureVector = new FeatureVector * [size = lines];
    vectorClass = new unsigned int [lines];

    // read in data
    instream->open(infile->c_str());
    for(i = 0; i < lines; i++)
    {
        instream->getline(line, MAXLINELEN);
        // swap classes here
```

```
      vectorClass[i] = atoi(line + featureVectorLength);

      featureVector[i] = new FeatureVector(featureVectorLength);
      j = featureVectorLength;
      while(j--)
      {
         line[j + 1] = '\0';
         featureVector[i]->value[j] = atoi(line + j);
      }
   }

   instream->close();
   delete instream;
}

// destructor
DataSet::~DataSet(void)
{
   register unsigned int i = size;

   while(i--)
      delete featureVector[i];
   delete [] vectorClass;
   delete [] featureVector;

}

// save to stream
void DataSet::save(FILE *outputStream)
{
   register unsigned int i;

   fprintf(outputStream, \
         "%-3d %-5d\n", \
         size, \
         vectorLength \
         );

   for(i = 0; i < size; i++)
      fprintf(outputStream, "%-1d ", vectorClass[i]);
   fprintf(outputStream, "\n");

   for(i = 0; i < size; i++)
      featureVector[i]->save(outputStream);

   fflush(outputStream);
}

// copy sample from one dataset to another
```

```
// dataSet - dataset to copy from
// fromIndex - index of sample in from data set
// toIndex - index of sample to copy to
void DataSet::copySample(DataSet *dataSet, const unsigned int fromIndex, \
      const unsigned int toIndex)
{
   register unsigned int i = dataSet->vectorLength;

   featureVector[toIndex]->length = dataSet->featureVector[fromIndex]->length;
   while(i--)
      featureVector[toIndex]->value[i] = \
            dataSet->featureVector[fromIndex]->value[i];
   vectorClass[toIndex] = dataSet->vectorClass[fromIndex];
}

// swap samples between datasets
// dataSet - dataset to copy from
// fromIndex - index of sample in from data set
// toIndex - index of sample to copy to
void DataSet::swapSample(DataSet *dataSet, const unsigned int fromIndex, \
      const unsigned int toIndex)
{
   register unsigned int i, tmpLength, tmpVectorClass;

   tmpLength = featureVector[toIndex]->length;
   tmpVectorClass = vectorClass[toIndex];
   unsigned int *tmpValue = new unsigned int [tmpLength];
   i = tmpLength;
   while(i--)
      tmpValue[i] = featureVector[toIndex]->value[i];

   copySample(dataSet, fromIndex, toIndex);
   vectorLength = dataSet->vectorLength;

   dataSet->featureVector[fromIndex]->length = tmpLength;
   dataSet->vectorClass[fromIndex] = tmpVectorClass;
   i = tmpLength;
   while(i--)
      dataSet->featureVector[fromIndex]->value[i] = tmpValue[i];

   delete [] tmpValue;
}

// split dataset
void DataSet::split(DataSet *dataSet1, DataSet *dataSet2, \
      const unsigned int dataSet1Size)
{
   register unsigned int i, index;
   static unsigned int *used = new unsigned int [size] = {0};
```

```
    dataSet1->size = dataSet1Size;
    dataSet1->vectorLength = vectorLength;
    dataSet2->size = size - dataSet1Size;
    dataSet2->vectorLength = vectorLength;

    // create random training set
    i = dataSet1Size;
    while(i--)
    {
        do
            index = int((double(rand()) * size) / double(RAND_MAX + 1.0));
        while(used[index]);
        dataSet1->copySample(this, index, i);
        used[index] = 1;
    }

    i = size;
    index = 0;
    while(i--)
    {
        if(!used[i])
            dataSet2->copySample(this, i, index++);
        used[i] = 0;
    }

    dataSet1->randomise();
    dataSet2->randomise();
}

// randomise dataset
void DataSet::randomise(void)
{
    register unsigned int i = size;

    while(i--)
        swapSample(this, int((double(rand()) * size) / double(RAND_MAX + 1.0)), \
                int((double(rand()) * size) / double(RAND_MAX + 1.0)));
}

//
// crossvalidation set class public methods
//

// constructor - create crossvalidation set
// dataSet - dataset to create from
// cvsets - num of crossvalidation sets to create
CrossvalidationSet::CrossvalidationSet(DataSet *dataSet, \
        const unsigned int cvsets)
```

```
{
  register unsigned int i, oddSize, evenSize, testStart, testEnd;
  register unsigned int dataIndex, testIndex, trainingIndex;

  this->cvsets = cvsets;
  evenSize = dataSet->size / cvsets;
  oddSize = evenSize + dataSet->size - evenSize * cvsets;

  trainingSet = new DataSet * [cvsets];
  testSet = new DataSet * [cvsets];

  if(cvsets == 1)
  {
    trainingSet[0] = new DataSet(dataSet->size, dataSet->vectorLength);
    testSet[0] = new DataSet(dataSet->size, dataSet->vectorLength);
    dataIndex = dataSet->size;
    while(dataIndex--)
    {
      trainingSet[0]->featureVector[dataIndex] = new FeatureVector(\
          dataSet->vectorLength);
      trainingSet[0]->copySample(dataSet, dataIndex, dataIndex);
      testSet[0]->featureVector[dataIndex] = new FeatureVector(\
          dataSet->vectorLength);
      testSet[0]->copySample(dataSet, dataIndex, dataIndex);
    }
  }
  else
  {
    trainingSet[0] = new DataSet(dataSet->size - oddSize, \
        dataSet->vectorLength);
    testSet[0] = new DataSet(oddSize, dataSet->vectorLength);
    testIndex = trainingIndex = dataIndex = 0;
    while(dataIndex < oddSize)
    {
      testSet[0]->featureVector[testIndex] = new FeatureVector(\
          dataSet->vectorLength);
      testSet[0]->copySample(dataSet, dataIndex++, \
          testIndex++);
    }
    while(dataIndex < dataSet->size)
    {
      trainingSet[0]->featureVector[trainingIndex] = new FeatureVector(\
          dataSet->vectorLength);
      trainingSet[0]->copySample(dataSet, dataIndex++, \
          trainingIndex++);
    }
  }

  for(i = 1; i < cvsets; i++)
```

```
  {
    trainingSet[i] = new DataSet(dataSet->size - evenSize, \
        dataSet->vectorLength);
    testSet[i] = new DataSet(evenSize, dataSet->vectorLength);
    testStart = oddSize + (i - 1) * evenSize;
    testEnd = testStart + evenSize;
    testIndex = trainingIndex = dataIndex = 0;
    while(dataIndex < testStart)
    {
      trainingSet[i]->featureVector[trainingIndex] = new FeatureVector(\
          dataSet->vectorLength);
      trainingSet[i]->copySample(dataSet, dataIndex++, \
          trainingIndex++);
    }
    while(dataIndex < testEnd)
    {
      testSet[i]->featureVector[testIndex] = new FeatureVector(\
          dataSet->vectorLength);
      testSet[i]->copySample(dataSet, dataIndex++, \
          testIndex++);
    }
    while(dataIndex < dataSet->size)
    {
      trainingSet[i]->featureVector[trainingIndex] = new FeatureVector(\
          dataSet->vectorLength);
      trainingSet[i]->copySample(dataSet, dataIndex++, \
          trainingIndex++);
    }
  }

  randomise();
}

// randomise order of feature vectors in crossvalidation set
void CrossvalidationSet::randomise(void)
{
  register unsigned int i, j, index1, index2;
  register short unsigned int flag, redo;
  DataSet *set1, *set2;

  i = cvsets * (trainingSet[0]->size + testSet[0]->size);
  while(i--)
  {
    if(drand48() < 0.5)
      set1 = trainingSet[int((double(rand()) * cvsets) / \
          double(RAND_MAX + 1.0))];
    else
      set1 = testSet[int((double(rand()) * cvsets) / \
          double(RAND_MAX + 1.0))];
```

```
        if(drand48() < 0.5)
            set2 = trainingSet[int((double(rand()) * cvsets) / \
                    double(RAND_MAX + 1.0))];
        else
            set2 = testSet[int((double(rand()) * cvsets) / \
                    double(RAND_MAX + 1.0))];

        index1 = int((double(rand()) * set1->size) / \
                    double(RAND_MAX + 1.0));
        index2 = int((double(rand()) * set2->size) / \
                    double(RAND_MAX + 1.0));
        set1->swapSample(set2, index2, index1);
    }

    redo = 0;
    i = cvsets;
    while(i--)
    {
        flag = trainingSet[i]->vectorClass[0];
        j = trainingSet[i]->size;
        while(--j)
            if(flag != trainingSet[i]->vectorClass[j])
                break;
        if(!j)
        {
            redo = 1;
            break;
        }
    }

    if(redo)
        randomise();
}
```

B.2.9 EvolutionaryAlgorithm.h

```
/****************************************************************************/
/* artificial immune system concept learner v1.0                            */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                     */
/* released under the gnu general public license                            */
/****************************************************************************/

/****************************************************************************/
/* cooperative coevolutionary algorithm routines                            */
/****************************************************************************/

#ifndef EVOLUTIONARYALGORITHM_H
#define EVOLUTIONARYALGORITHM_H

// headers
#include "Classifier.h"
#include <stdio.h>

//
// genome class
//
class Genome
{
   public:
      Genome(const unsigned int length);
      ~Genome(void);

      unsigned int size;  // num of genes
      unsigned int *locus;  // loci values
      unsigned int type;  // genome type
      double mutationProbability;
      double crossoverProbability;
      double fitness, scaledFitness;
      unsigned int thresholdLength, patternLength;
      double generalityBias;
      double typeBias;

      void copyGenome(Genome *genome);
      void uniformCrossover(Genome *genome1, Genome *genome2);
      void mutateBinary(void);
      void randomiseBinary(void);
      void save(FILE *outputStream);
      void setDetector(Detector *detector);
      void show(void) { save(stdout); };
};

//
// species class
```

```
//
class Species
{
  public:
    Species(const unsigned int speciesSize, const unsigned int genomeLength);
    ~Species(void);

    unsigned int speciesSize;
    Genome **genome;  // genomes in species
    unsigned int fittestIndividual;
    double speciesScaledFitnessSum;
    double meanSpeciesFitness;

    Genome *FPSelection(void);
    void randomise(void);
    void save(FILE *outputStream);
    void copySpecies(Species *species);
    void show(void) { save(stdout); };
};

#endif
```

B.2.10 EvolutionaryAlgorithm.cpp

```
/****************************************************************************/
/* artificial immune system concept learner v1.0                            */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                     */
/* released under the gnu general public license                            */
/****************************************************************************/

/****************************************************************************/
/* cooperative coevolutionary algorithm routines                            */
/****************************************************************************/

// headers
#include "EvolutionaryAlgorithm.h"
#include <stdlib.h>

//
// genome class public methods
//

// constructor - create genome
// length - length of genome
Genome::Genome(const unsigned int length)
{
   thresholdLength = 8;
   patternLength = length;
   size = thresholdLength + 2 * patternLength;
   locus = new unsigned int [size];
   mutationProbability = 2.0 / double(size);
   crossoverProbability = 0.6;
   generalityBias = typeBias = 0.5;
   fitness = 0.0;
   type = 0;
}

// destructor
Genome::~Genome(void)
{
   delete [] locus;
}

// copy genome
// genome - genome to copy from
void Genome::copyGenome(Genome *genome)
{
   register unsigned int i = size;
   register unsigned int *from = genome->locus;
   register unsigned int *to = locus;
```

```
    while(i--)
            to[i] = from[i];
    mutationProbability = genome->mutationProbability;
    crossoverProbability = genome->crossoverProbability;
    fitness = genome->fitness;
    scaledFitness = genome->scaledFitness;
    generalityBias = genome->generalityBias;
    size = genome->size;
    patternLength = genome->patternLength;
    thresholdLength = genome->thresholdLength;
    type = genome->type;
}

// uniform crossover
// genome1, genome2 - genome to create from
void Genome::uniformCrossover(Genome *genome1, Genome *genome2)
{
    register unsigned int i = size;
    register unsigned int *from1 = genome1->locus;
    register unsigned int *from2 = genome2->locus;
    register unsigned int *to = locus;
    register double cp = crossoverProbability;

    while(i--)
    {
        if(drand48() < cp)
            to[i] = from1[i];
        else
            to[i] = from2[i];
    }
    if(drand48() < cp)
        type = genome1->type;
    else
        type = genome2->type;
}

// binary bit-flip mutation
void Genome::mutateBinary(void)
{
    register unsigned int i = size;
    register unsigned int *loci = locus;
    register double mp = mutationProbability;

    while(i--)
        if(drand48() < mp)
            loci[i] = 1 - loci[i];
    if(drand48() < mp)
            type = 1 - type;
}
```

```
// randomise binary genome
void Genome::randomiseBinary(void)
{
   register unsigned int index, i;

   index = 0;

   i = thresholdLength;
   while(i--)
      locus[index++] = int((double(rand()) * 2.0) / double(RAND_MAX + 1.0));
   i = patternLength;
   while(i--)
      locus[index++] = int((double(rand()) * 2.0) / double(RAND_MAX + 1.0));
   i = patternLength;
   while(i--)
      if(drand48() < generalityBias)
         locus[index++] = 0;
      else
         locus[index++] = 1;
   if(drand48() < typeBias)
      type = SELF;
   else
      type = NONSELF;
}

// save to stream
void Genome::save(FILE *outputStream)
{
   register unsigned int i;
   Detector *detector = new Detector(patternLength);

   fprintf(outputStream, \
         "%-3d %-3d %-3d %-1d %-10f %-10f %-10f %-10f %-10f %-10f\n", \
         size, \
         thresholdLength, \
         patternLength, \
         type, \
         fitness, \
         scaledFitness, \
         mutationProbability, \
         crossoverProbability, \
         generalityBias, \
         typeBias \
         );

   for(i = 0; i < size; i++)
      fprintf(outputStream, "%-2d ", locus[i]);
   fprintf(outputStream, "\n");
```

```
    setDetector(detector);
    detector->save(outputStream);

    delete detector;

    fflush(outputStream);
}

// create detector from genome
// detector - detector to create
void Genome::setDetector(Detector *detector)
{
    register unsigned int i, loci = 0, sum, lastLoci;

    // set activation threshold
    // gray coding for threshold gene
    sum = lastLoci = locus[loci++];
    while(loci < thresholdLength)
    {
        sum = (sum << 1) | (lastLoci ^ locus[loci]);
        lastLoci = locus[loci++];
    }
    detector->threshold = double(sum) / 255.0; // !!!!!!!!!!!!!!!!!!!!!
    for(i = 0; i < patternLength; i++)
        detector->value[i] = locus[loci++];
    for(i = 0; i < patternLength; i++)
        if(!locus[loci++])
            detector->value[i] = MASKVALUE;
    detector->type = type;
    detector->length = patternLength;
}

//
// species class public methods
//

// constructor - create species
// genomeLength - length of individuals' genomes
Species::Species(const unsigned int speciesSize, \
        const unsigned int genomeLength)
{
    register unsigned int i = speciesSize;

    this->speciesSize = speciesSize;
    fittestIndividual = 0;
    speciesScaledFitnessSum = meanSpeciesFitness = 0.0;
    genome = new Genome * [speciesSize];
    while(i--)
```

```
      genome[i] = new Genome(genomeLength);
}

// destructor
Species::~Species(void)
{
   register unsigned int i = speciesSize;

   while(i--)
      delete genome[i];
   delete genome;
}

// fitness proportionate selection
Genome *Species::FPSelection(void)
{
   register unsigned int i = 0;
   register double dtmp1, dtmp2;

   dtmp1 = drand48() * speciesScaledFitnessSum;
   dtmp2 = 0.0;
   while((i < speciesSize) && ((dtmp2 = dtmp2 + genome[i]->scaledFitness) \
         < dtmp1))
      i++;
   return((i < speciesSize) ? genome[i] : genome[i - 1]);
}

// randomise all species
void Species::randomise(void)
{
   register unsigned int i = speciesSize;

   while(i--)
      genome[i]->randomiseBinary();
}

// save to stream
void Species::save(FILE *outputStream)
{
   fprintf(outputStream, \
         "%-4d %-4d %-5.10f %-.10f\n", \
         speciesSize, \
         fittestIndividual, \
         speciesScaledFitnessSum, \
         meanSpeciesFitness \
         );

   genome[fittestIndividual]->save(outputStream);
```

```
    fflush(outputStream);
}

// copy species
// species - species to copy from
void Species::copySpecies(Species *species)
{
    register unsigned int i = species->speciesSize;

    speciesSize = i;
    while(i--)
        genome[i]->copyGenome(species->genome[i]);
    fittestIndividual = species->fittestIndividual;
    speciesScaledFitnessSum = species->speciesScaledFitnessSum;
    meanSpeciesFitness = species->meanSpeciesFitness;
}
```

B.2.11 FeatureExtractor.h

```
/*****************************************************************************/
/* artificial immune system concept learner v1.0                             */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                      */
/* released under the gnu general public license                             */
/*****************************************************************************/

/*****************************************************************************/
/* feature extractor routines                                                */
/*****************************************************************************/

#ifndef FEATUREEXTRACTOR_H
#define FEATUREEXTRACTOR_H

// headers
#include <string>
#include "DataSet.h"
#include <stdio.h>

// constants
#define FIELD_SEPERATOR     '|'  // seperator for file input

#define SELF                0  // self
#define NONSELF             1  // nonself

#define MAX_VECTOR_LENGTH  256  // default max vector len
#define MAX_WORDS          10000  // default max words
#define MAX_DOCUMENTS      200  // default max documents

//
// feature extractor class
//
class FeatureExtractor
{
  public:
    FeatureExtractor(string *infile, \
        const unsigned int maxVectorLength = MAX_VECTOR_LENGTH, \
        const unsigned int maxWords = MAX_WORDS, \
        const unsigned int maxDocuments = MAX_DOCUMENTS);
    ~FeatureExtractor(void);

    unsigned int numDocuments;
    unsigned int numWords;
    unsigned int vectorLength;
    unsigned int maxVectorLength, maxWords, maxDocuments;
    unsigned int numSelf, numNonself;
    unsigned int numTrainingExamples, numTrainingSelf, numTrainingNonself;
    string **documentFilename;
    string **documentName;
```

```
    unsigned int *documentClass;
    DataSet *dataSet;
    DataSet *trainingSet;
    DataSet *testSet;

    string **word;
    unsigned int **wordTable;
    char **stopList;
    unsigned int stopListLength;
    unsigned int *trainingVector;
    unsigned int *trainingWords;

    // data set statistics
    unsigned int *present, *absent;
    double pSelf, pNonself;
    double *pPresent, *pAbsent;
    unsigned int *presentSelf, *presentNonself;
    unsigned int *absentSelf, *absentNonself;
    double *pSelfPresent, *pNonselfPresent, *pSelfAbsent, *pNonselfAbsent;
    double IS;  // I(S)
    double *EIG;  // E(w,S)
    short unsigned int *flag;
    unsigned int **trainingSetIndex;

    void createDataSet(const unsigned int vectorLength, \
          const unsigned int numTrainingExamples);
    void createDataSet(const unsigned int vectorLength, \
          const unsigned int numTrainingExamples, \
          unsigned int *trainingExamples);
    void save(FILE *outputStream);
    void show(void) { save(stdout); };
    void createTrainingSets(const unsigned int cvSets);
    void createFromTrainingSet(const unsigned int set);

  private:
    int onStopList(char *word);
    void calculateStatistics(void);
};

#endif
```

B.2.12 FeatureExtractor.cpp

```
/**************************************************************************/
/* artificial immune system concept learner v1.0                          */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                   */
/* released under the gnu general public license                          */
/**************************************************************************/

/**************************************************************************/
/* feature extractor routines                                             */
/**************************************************************************/

// headers
#include "FeatureExtractor.h"
#include <fstream>
#include <stdio.h>
#include <math.h>

// constants
#define MAXLINELEN   1000  // max input line length

// the default stop list
#define DEFSTOPLISTLEN     80
static char *defaultStopList[DEFSTOPLISTLEN] = {"WWW", "HTML", "HTTP", "GIF",
"EDU", "AND", "HREF", "THE", "IMG", "SRC", "FOR", "FONT", "COM", "ALIGN", "ALT",
"SIZE", "INDEX", "HTM", "TITLE", "GOPHER", "ORG", "NAME", "THIS", "WEB", "YOU",
"HOME", "ABOUT", "INTERNET", "WIDTH", "PAGE", "FTP", "BODY", "ARE", "LIST",
"NET", "HEIGHT", "LINKS", "NEWS", "FROM", "HEAD", "STRONG", "WELCOME", "WITH",
"TOP", "MAILTO", "YOUR", "GIFS", "BOTTOM", "MAIL", "CGI", "THAT", "BIN", "ALL",
"CENTER", "WUSTL", "GDB", "GOV", "OTHER", "ANY", "HAS", "NOT", "TOC", "GNN",
"WIC", "SERVER", "AVAILABLE", "IBC", "ADDRESS", "INFORMATION", "HERE", "CAN",
"WHAT", "MORE", "OUR", "WILL", "HAVE", "COMMENTS", "WHO", "PLEASE", "ALSO"};

// quick log and abs macros
#define log2(a)          (((a) == 0.0) ? (0.0) : (log(a) / log(2.0)))
#define qfabs(a)         (((a) < 0.0) ? (1.0 * (a)): (a))

//
// feature extractor class public methods
//

// constructor - create feature extractor
// infile - input file
FeatureExtractor::FeatureExtractor(string *infile, \
      const unsigned int maxVectorLength, const unsigned int maxWords, \
            const unsigned int maxDocuments)
{
   register unsigned int i, j, lines, dirend;
   char line[MAXLINELEN];
```

```
register char c;
register short unsigned int flag;
ifstream *instream = new ifstream();;

// count num of lines
instream->open(infile->c_str());
lines = 1;
while(instream->getline(line, MAXLINELEN))
  lines++;
instream->close();

// initialise memory and set defaults
numDocuments = lines;
numWords = vectorLength = numSelf = numNonself = 0;
this->maxVectorLength = maxVectorLength;
this->maxWords = maxWords;
this->maxDocuments = maxDocuments;
numTrainingExamples = numTrainingSelf = numTrainingNonself = 0;

documentFilename = new string * [numDocuments];
documentName = new string * [numDocuments];
documentClass = new unsigned int [numDocuments];
dataSet = new DataSet(numDocuments, maxVectorLength);
trainingSet = new DataSet(numDocuments, maxVectorLength);
testSet = new DataSet(numDocuments, maxVectorLength);

word = new string * [maxWords];
wordTable = new unsigned int * [maxDocuments];
trainingSetIndex = new unsigned int * [maxDocuments];
i = maxDocuments;
while(i--)
{
  wordTable[i] = new unsigned int [maxWords];
  trainingSetIndex[i] = new unsigned int [maxDocuments];
}
stopList = defaultStopList;
stopListLength = DEFSTOPLISTLEN;
trainingVector = new unsigned int [maxDocuments];
trainingWords = new unsigned int [maxWords];

present = new unsigned int [maxWords];
absent = new unsigned int [maxWords];
IS = pSelf = pNonself = 0.0;
pPresent = new double [maxWords];
pAbsent = new double [maxWords];
presentSelf = new unsigned int [maxWords];
presentNonself = new unsigned int [maxWords];
absentSelf = new unsigned int [maxWords];
absentNonself = new unsigned int [maxWords];
```

```
pSelfPresent = new double [maxWords];
pNonselfPresent = new double [maxWords];
pSelfAbsent = new double [maxWords];
pNonselfAbsent = new double [maxWords];
EIG = new double [maxWords];
this->flag = new short unsigned int [maxWords];

// find path of input files
dirend = infile->find_last_of('/') + 1;
cout << dirend <<"\n";

// read in file data
instream->open(infile->c_str());
for(i = 0; i < numDocuments; i++)
{
   instream->getline(line, MAXLINELEN, FIELD_SEPERATOR);
   documentFilename[i] = new string(*infile, 0, dirend);
   documentFilename[i]->append(line);
   instream->getline(line, MAXLINELEN, FIELD_SEPERATOR);
   switch(strcmp(line, "hot"))
   {
      case 0:
         documentClass[i] = SELF;
         numSelf++;
         break;

      default:
         documentClass[i] = NONSELF;
         numNonself++;
         break;
   }
   instream->getline(line, MAXLINELEN);
   documentName[i] = new string(line);
}
instream->close();

// create word table
numWords = 0;
for(i = 0; i < numDocuments; i++)
{
   instream->open(documentFilename[i]->c_str());
   flag = 1;
   while(flag)
   {
      while(((c = instream->get()) != EOF) && !isalpha(c))
         ;
      if(c == EOF)
         flag = 0;
      else
```

```
      {
        j = 0;
        while((c != EOF) && isalpha(c))
        {
          line[j++] = toupper(c);   // convert to upper case
          c = instream->get();
        }
        line[j] = '\0';
        if(onStopList(line))
          continue;
        for(j = 0; j < numWords; j++)
          if(strcmp(word[j]->c_str(), line) == 0)
          {
            wordTable[i][j]++;
            break;
          }
        if(j == numWords)
        {
          word[numWords] = new string(line);
          wordTable[i][numWords++]++;
        }
      }
    }
    instream->close();
  }

  delete instream;
}

// destructor
FeatureExtractor::~FeatureExtractor(void)
{
  register unsigned int i;

  delete dataSet;
  delete trainingSet;
  delete testSet;

  delete [] documentClass;
  delete [] trainingVector;
  delete [] trainingWords;

  delete [] present;
  delete [] absent;
  delete [] pPresent;
  delete [] pAbsent;
  delete [] presentSelf;
  delete [] presentNonself;
  delete [] absentSelf;
```

```
   delete [] absentNonself;
   delete [] pSelfPresent;
   delete [] pNonselfPresent;
   delete [] pSelfAbsent;
   delete [] pNonselfAbsent;
   delete [] EIG;
   delete [] flag;

   i = maxWords;
   while(i--)
      delete word[i];
   delete [] word;

   i = maxDocuments;
   while(i--)
   {
      delete documentName[i];
      delete documentFilename[i];
      delete [] wordTable[i];
   }
   delete [] wordTable;
   delete [] documentFilename;
   delete [] documentName;
}

// create data set
// vectorLength - length of feature vectors to extract
void FeatureExtractor::createDataSet(const unsigned int vectorLength, \
            const unsigned int numTrainingExamples)
{
   register unsigned int i, j, index;
   register short unsigned int flag;

   this->numTrainingExamples = numTrainingExamples;
   this->vectorLength = vectorLength;

   // create random training set
   numTrainingSelf = numTrainingNonself = 0;
   i = numTrainingExamples;
   while(i)
   {
      do
      {
         flag = 0;
         index = int((double(rand()) * numDocuments) / double(RAND_MAX + 1.0));
         j = numTrainingExamples - i;
         while(j--)
            if(trainingVector[j] == index)
            {
```

```
            flag = 1;
            continue;
         }
      } while(flag);
      trainingVector[numTrainingExamples - i] = index;
      if(documentClass[trainingVector[i]] == SELF)
         numTrainingSelf++;
      else
         numTrainingNonself++;
      i--;
   }

   calculateStatistics();
}

// create data set
// vectorLength - length of feature vectors to extract
// trainingExamples - indices of training examples to create dataset from
void FeatureExtractor::createDataSet(const unsigned int vectorLength, \
         const unsigned int numTrainingExamples, \
         unsigned int *trainingExamples)
{
   register unsigned int i;

   this->numTrainingExamples = numTrainingExamples;
   this->vectorLength = vectorLength;

   // create random training set
   numTrainingSelf = numTrainingNonself = 0;
   i = numTrainingExamples;
   while(i--)
   {
      trainingVector[i] = trainingExamples[i];
      if(documentClass[trainingVector[i]] == SELF)
         numTrainingSelf++;
      else
         numTrainingNonself++;
   }

   calculateStatistics();
}

// save to stream
void FeatureExtractor::save(FILE *outputStream)
{
   register unsigned int i, j;

   fprintf(outputStream, \
   "%-3d %-5d %-3d %-3d %-5d %-3d %-3d %-3d %-3d %-3d %-3d %-10f %-10f %-10f\n", \
```

```
        numDocuments, \
        numWords, \
        vectorLength, \
        maxVectorLength, \
        maxWords, \
        maxDocuments, \
        numSelf, \
        numNonself, \
        numTrainingExamples, \
        numTrainingSelf, \
        numTrainingNonself, \
        pSelf, \
        pNonself, \
        IS \
        );

for(i = 0; i < numDocuments; i++)
    fprintf(outputStream, \
          "%-3d %-30s %-30s %-1d\n", \
          i, \
          documentFilename[i]->c_str(), \
          documentName[i]->c_str(), \
          documentClass[i] \
          );
for(i = 0; i < numTrainingExamples; i++)
    fprintf(outputStream, "%-3d\n", trainingVector[i]);

for(i = 0; i < vectorLength; i++)
    fprintf(outputStream, \
          "%-3d %-5d\n", \
          i, \
          trainingWords[i] \
          );

for(i = 0; i < numWords; i++)
{
    fprintf(outputStream, \
          "%-3d %-20s %-.10f %-3d %-3d %-3d %-3d %-3d %-3d %-.10f %-.10f ", \
          i, \
          word[i]->c_str(), \
          EIG[i], \
          present[i], \
          absent[i], \
          presentSelf[i], \
          absentSelf[i], \
          presentNonself[i], \
          absentNonself[i], \
          pPresent[i], \
          pAbsent[i] \
```

```
            );
      fprintf(outputStream, \
            "%-.10f %-.10f %-.10f %-.10f\n", \
            pSelfPresent[i], \
            pSelfAbsent[i], \
            pNonselfPresent[i], \
            pNonselfAbsent[i] \
            );
   }

   for(i = 0; i < numDocuments; i++)
   {
      fprintf(outputStream, "%-5d ", i);
      for(j = 0; j < numWords; j++)
         fprintf(outputStream, " %-3d", wordTable[i][j]);
      fprintf(outputStream, "\n");
   }

   fprintf(outputStream, "%-5d", stopListLength);
   for(i = 0; i < stopListLength; i++)
      fprintf(outputStream, " %-20s", stopList[i]);
   fprintf(outputStream, "\n");

   fflush(outputStream);
}

// extract training sets
// cvSets - num of training sets to create
void FeatureExtractor::createTrainingSets(const unsigned int cvSets)
{
   register unsigned int i, index;
   register unsigned int testSetSize, trainingSetIndex, evenNumDocuments;
   register unsigned int toIndex, tmpValue;

   testSetSize = int(numDocuments / cvSets);
   evenNumDocuments = testSetSize * cvSets;
   numTrainingExamples = testSetSize * (cvSets - 1);

   i = evenNumDocuments;
   while(i--)
      trainingVector[i] = i;

   i = evenNumDocuments;
   while(i--)
   {
      toIndex = int((double(rand()) * evenNumDocuments) / \
            double(RAND_MAX + 1.0));
      tmpValue = trainingVector[i];
      trainingVector[i] = trainingVector[toIndex];
```

```
      trainingVector[toIndex] = tmpValue;
   }

   i = cvSets;
   while(i--)
   {
      trainingSetIndex = index = 0;
      toIndex = i * testSetSize;
      while(toIndex--)
         this->trainingSetIndex[i][trainingSetIndex++] = \
               trainingVector[index++];
      index += testSetSize;
      toIndex = (cvSets - i - 1) * testSetSize;
      while(toIndex--)
         this->trainingSetIndex[i][trainingSetIndex++] = \
               trainingVector[index++];
   }
}

// create dataset from training set
// set - training set index
void FeatureExtractor::createFromTrainingSet(const unsigned int set)
{
   createDataSet(vectorLength, numTrainingExamples, trainingSetIndex[set]);
}

//
// feature extractor class private methods
//

// check if word on stop list
int FeatureExtractor::onStopList(char *word)
{
   register unsigned int i = stopListLength;

   while(i--)
      if(strcmp(stopList[i], word) == 0)
         return(1);
   return(0);
}

// calculate statistics for dataset
void FeatureExtractor::calculateStatistics(void)
{
   register unsigned int i, j, k;
   register short unsigned int flag;
   register double currentBestEIG, lastBestEIG;
   register unsigned int currentBestEIGIndex, trainingIndex, testIndex;
```

```
// calculate statistics
pSelf = double(numTrainingSelf) / double(numTrainingExamples);
pNonself = double(numTrainingNonself) / double(numTrainingExamples);
IS = -1.0 * (pSelf * log2(pSelf) + pNonself * log2(pNonself));

// reset statistics
i = numWords;
while(i--)
{
   present[i] = absent[i] = presentSelf[i] = absentSelf[i] = \
         presentNonself[i] = absentNonself[i] = this->flag[i] = 0;
   EIG[i] = pPresent[i] = pAbsent[i] = pSelfPresent[i] = \
         pNonselfPresent[i] = pSelfAbsent[i] = pNonselfAbsent[i] = 0.0;
}

for(i = 0; i < numTrainingExamples; i++)
   for(j = 0; j < numWords; j++)
      if(documentClass[trainingVector[i]] == SELF)
      {
         if(wordTable[trainingVector[i]][j])
         {
            presentSelf[j]++;
            present[j]++;
         }
         else
         {
            absentSelf[j]++;
            absent[j]++;
         }
      }
      else
      {
         if(wordTable[trainingVector[i]][j])
         {
            presentNonself[j]++;
            present[j]++;
         }
         else
         {
            absentNonself[j]++;
            absent[j]++;
         }
      }

for(i = 0; i < numTrainingExamples; i++)
   for(j = 0; j < numWords; j++)
   {
      if(present[j])
      {
```

```
            pPresent[j] = double(present[j]) / double(numTrainingExamples);
            pSelfPresent[j] = double(presentSelf[j]) / double(present[j]);
            pNonselfPresent[j] = double(presentNonself[j]) / double(present[j]);
        }
        if(absent[j])
        {
            pAbsent[j] = double(absent[j]) / double(numTrainingExamples);
            pSelfAbsent[j] = double(absentSelf[j]) / double(absent[j]);
            pNonselfAbsent[j] = double(absentNonself[j]) / double(absent[j]);
        }

        EIG[j] = IS + pPresent[j] * (pSelfPresent[j] * \
                log2(pSelfPresent[j]) + \
                pNonselfPresent[j] * log2(pNonselfPresent[j])) + \
                pAbsent[j] * (pSelfAbsent[j] * log2(pSelfAbsent[j]) + \
                pNonselfAbsent[j] * log2(pNonselfAbsent[j]));
    }

lastBestEIG = 200.0;
currentBestEIGIndex = 0;
for(i = 0; i < vectorLength; i++)
{
    currentBestEIG = -100.0;
    for(j = 0; j < numWords; j++)
        if((EIG[j] > currentBestEIG) \
                && (EIG[j] <= lastBestEIG) \
                && !this->flag[j])
        {
            flag = 0;
            for(k = 0; k < numTrainingExamples; k++)
                if(wordTable[trainingVector[k]][j] != 0)
                {
                    flag = 1;
                    break;
                }
            if(flag)
            {
                currentBestEIG = EIG[j];
                currentBestEIGIndex = j;
            }
        }
    // add to list
    trainingWords[i] = currentBestEIGIndex;
    this->flag[currentBestEIGIndex] = 1;
    lastBestEIG = currentBestEIG;
}

trainingSet->size = numTrainingExamples;
testSet->size = numDocuments - numTrainingExamples;
```

```
    trainingSet->vectorLength = testSet->vectorLength = vectorLength;
    dataSet->vectorLength = vectorLength;
    trainingIndex = testIndex = 0;
    for(i = 0; i < numDocuments; i++)
    {
        for(j = 0; j < vectorLength; j++)
            if(wordTable[i][trainingWords[j]])
                dataSet->featureVector[i]->value[j] = 1;
            else
                dataSet->featureVector[i]->value[j] = 0;
        dataSet->vectorClass[i] = documentClass[i];
        dataSet->featureVector[i]->length = vectorLength;

        flag = 0;
        j = numTrainingExamples;
        while(j--)
            if(i == trainingVector[j])
            {
                flag = 1;
                break;
            }
        if(flag)
        {
            for(j = 0; j < vectorLength; j++)
                if(wordTable[i][trainingWords[j]])
                    trainingSet->featureVector[trainingIndex]->value[j] = 1;
                else
                    trainingSet->featureVector[trainingIndex]->value[j] = 0;
            trainingSet->vectorClass[trainingIndex] = documentClass[i];
            trainingSet->featureVector[trainingIndex]->length = vectorLength;
            trainingIndex++;
        }
        else
        {
            for(j = 0; j < vectorLength; j++)
                if(wordTable[i][trainingWords[j]])
                    testSet->featureVector[testIndex]->value[j] = 1;
                else
                    testSet->featureVector[testIndex]->value[j] = 0;
            testSet->vectorClass[testIndex] = documentClass[i];
            testSet->featureVector[testIndex]->length = vectorLength;
            testIndex++;
        }
    }
}
```

B.2.13 NaiveBayesianClassifier.h

```
/****************************************************************************/
/* artificial immune system concept learner v1.0                            */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                     */
/* released under the gnu general public license                            */
/****************************************************************************/

/****************************************************************************/
/* naive bayesian classifier routines                                       */
/****************************************************************************/

#ifndef NAIVEBAYESIANCLASSIFIER_H
#define NAIVEBAYESIANCLASSIFIER_H

// headers
#include "DataSet.h"

//
// naive bayesian classifier class
//
class NaiveBayesianClassifier
{
   public:
      NaiveBayesianClassifier(const unsigned int classifierSize);
      ~NaiveBayesianClassifier(void);

      unsigned int classifierSize;

      unsigned int totself, totnonself;  // total self/nonself
      double pself, pnonself;  // probabilities
      unsigned int *present, *absent;
      unsigned int *presself, *absself;
      unsigned int *presnonself, *absnonself;
      double *pselfpres, *pnonselfpres, *pselfabs, *pnonselfabs;
      double *ppres, *pabs;

      void train(DataSet *dataSet);
      double test(DataSet *dataSet);
};

#endif
```

B.2.14 NaiveBayesianClassifier.cpp

```
/*****************************************************************************/
/* artificial immune system concept learner v1.0                             */
/* copyright (c) 2002 jamie twycross, jamie@milieu3.net                      */
/* released under the gnu general public license                            */
/*****************************************************************************/

/*****************************************************************************/
/* naive bayesian classifier routines                                        */
/*****************************************************************************/

// headers
#include "NaiveBayesianClassifier.h"
#include "Classifier.h"

// constructure - create naive bayesian classifier
NaiveBayesianClassifier::NaiveBayesianClassifier( \
      const unsigned int classifierSize)
{
   this->classifierSize = classifierSize;

   pself = pnonself = 0.0;
   totself = totnonself = 0;
   pselfpres = new double [classifierSize];
   pselfabs = new double [classifierSize];
   pnonselfpres = new double [classifierSize];
   pnonselfabs = new double [classifierSize];

   present = new unsigned int [classifierSize];
   absent = new unsigned int [classifierSize];
   presself = new unsigned int [classifierSize];
   presnonself = new unsigned int [classifierSize];
   absself = new unsigned int [classifierSize];
   absnonself = new unsigned int [classifierSize];
   ppres = new double [classifierSize];
   pabs = new double [classifierSize];
}

// destructor
NaiveBayesianClassifier::~NaiveBayesianClassifier(void)
{
   delete [] pselfpres;
   delete [] pselfabs;
   delete [] pnonselfpres;
   delete [] pnonselfabs;

   delete [] present;
   delete [] absent;
```

```
   delete [] presself;
   delete [] presnonself;
   delete [] absself;
   delete [] absnonself;
   delete [] ppres;
   delete [] pabs;
}

// train naive bayesian classifier
// dataSet - dataset to train on
void NaiveBayesianClassifier::train(DataSet *dataSet)
{
   register unsigned int i, j;

   totself = totnonself = 0;
   i = classifierSize;
   while(i--)
   {
      present[i] = absent[i] = absself[i] = absnonself[i] = presself[i] = \
            presnonself[i] = 0;
   }

   i = dataSet->size;
   while(i--)
   {
      switch(dataSet->vectorClass[i])
      {
         case SELF:
            totself++;
            break;

         default:
            totnonself++;
            break;
      }
   }

   pself = double(totself) / double(dataSet->size);
   pnonself = double(totnonself) / double(dataSet->size);

   i = dataSet->size;
   while(i--)
   {
      j = classifierSize;
      while(j--)
      {
         switch(dataSet->featureVector[i]->value[j])
         {
            case 0:
```

```
            switch(dataSet->vectorClass[i])
            {
               case SELF:
                  absself[j]++;
                  break;

               default:
                  absnonself[j]++;
                  break;
            }
            present[j]++;
            break;

         case 1:
            switch(dataSet->vectorClass[i])
            {
               case SELF:
                  presself[j]++;
                  break;

               default:
                  presnonself[j]++;
                  break;
            }
            absent[j]++;
            break;

         default:
            break;
      }
   }
}

i = classifierSize;
while(i--)
{
   ppres[i] = double(present[i]) / double(dataSet->size);
   pabs[i] = double(absent[i]) / double(dataSet->size);
   if(present[i])
   {
      pselfpres[i] = double(presself[i]) / double(present[i]);
      pnonselfpres[i] = double(presnonself[i]) / double(present[i]);
   }
   if(absent[i])
   {
      pselfabs[i] = double(absself[i]) / double(absent[i]);
      pnonselfabs[i] = double(absnonself[i]) / double(absent[i]);
   }
}
```

```
}

// test naive bayesian classifier
// dataSet - dataset to test on
// returns predictive accuracy
double NaiveBayesianClassifier::test(DataSet *dataSet)
{
   register unsigned int i = dataSet->size, j;
   register unsigned int correct = 0;
   register double selfval, nonselfval;

   correct = 0;
   while(i--)
   {
      selfval = pself;
      nonselfval = pnonself;
      j = classifierSize;
      while(j--)
      {
         selfval *= ((dataSet->featureVector[i]->value[j]) ? (pselfpres[j]) : \
               (pselfabs[j]));
         nonselfval *= ((dataSet->featureVector[i]->value[j]) ? \
               (pnonselfpres[j]) : (pnonselfabs[j]));
      }
      if(selfval > nonselfval)
      {
         if(dataSet->vectorClass[i] == SELF)
            correct++;
      }
      else
         if(dataSet->vectorClass[i] == NONSELF)
            correct++;
   }

   return(double(correct) / double(dataSet->size));
}
```