Computational Analysis of Genomes

Matthew R. Pocock

This dissertation is submitted for the degree of Doctor of Philosophy.

April 2003

Supervisors: Dr T. J. P. Hubbard, Dr N. Goldman

The Sanger Centre, Cambridge; Darwin College, Cambridge

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

The work in this thesis has not been submitted in whole, or in part, for a degree, diploma, or any other qualification at any other university.

Matthew R. Pocock,

April 2003, Cambridge, United Kingdom

Dedication

I would like to thank all of those who have supported me through the process of producing this dissertation. Particular thanks must go to Tim Hubbard, who has been a source of great help and provided direction where needed without smothering me with micro-management. Nick Goldman and Ed Griffiths have both been valuable sounding boards throughout. Thomas Down has been my staunchest ally as we developed BioJava and also the DAS protocol from embryonic beginnings to the well-respected projects they are now. It would be unfair not to thank all of those who have helped with BioJava, as coders, testers and users. In particular, a mention must go to Chris Dagdigian for managing the hardware. I have enjoyed my time here, and this is in no small part due to the friendliness of those I meet daily in the Sanger Centre, the EBI and from the Ensembl project. Lastly, I must dedicate this work with heartfelt gratitude to Caroline. Without her love all academic achievements would be worthless.

Fate is unmoved by one's pitiful hopes; what changes, bowing to fate, is what one hopes for.

(Liza Dalby, The Tale of Murasaki p239)

Abstract

Recently we have been blessed with a simultaneous rise in the volume of biological data and the power of computers. This has necessarily led to the emergence of the field of Bioinformatics, where the study of entire genomes rather than individual genes is the norm.

This dissertation describes the development and application of the software framework BioJava, designed from the outset to provide a strong foundation for the implementation of different machine learning algorithms. BioJava allows genomic size datasets to be efficiently manipulated in a range of hardware environments.

A variety of supervised and unsupervised learning techniques were applied to data sets on the scale of whole genomes taking advantage of the BioJava framework.

Firstly, unsupervised learning was used to look for underlying structure in the genome sequence of whole Malaria chromosomes. Time-reversible 1st order Hidden Markov Models (HMMs) learned signals based on sequence composition that appear to correlate closely with biological units, such as exons, introns, repeats and non-coding genomic regions. This demonstrates the ability of unsupervised methods to discover biologically meaningful information within genomic sequence.

Secondly, supervised learning was used to develop a regression method able to predict recombination rate within human chromosomes. Support Vector Machines (SVMs) using suffix tree kernels were trained on human chromosome 22 sequence and were able to learn a signal reproducibly, although it was not clear how well this models recombination rate.

Finally, supervised learning was used to develop a classification method able to detect subtle signals in noisy and small sets of micro-array expression data. A Bayesian technique for training linear models was applied to learn sparse models. These were able to distinguish between tumour samples that had been treated with a drug and those that had not. The models produced by this method can be readily interpreted in terms of individual genes, and in this case made good biological sense.

This dissertation illustrates how a framework of modular and reusable software components can be used together with advances in artificial intelligence to help us interpret the data flowing from high throughput projects in the post genomic era.

Table of Contents

Dedication ii

Abstract iii

Table of Contents v

Table of Figures ix

List of Tables xi

Table of Equations xii

Chapter 1 Introduction 1

1.1 Existing Software Development Frameworks for Bioinformatics 2

1.1.1 The NCBI Toolkit 3

1.1.2 Bioperl 4

1.1.3 EMBOSS 5

1.2 BioJava 6

1.3 Machine Learning 8

1.3.1 Clustering, Classification and Regression for Single Items 9

1.3.2 Signal Analysis with Hidden Markov Models 18

1.4 Implementation and Use of BioJava 24

Chapter 2 The BioJava Core Interfaces 24

2.1 Java as a Language for Bioinformatics 24

2.2 Nested Exceptions and Assertions 24

2.3 Changeability 24

2.4 Symbols, Alphabets and SymbolList 24

2.5 Locations, Sequences and Features 24

2.6 Probability Distributions and Hidden Markov Models 24

2.7 Query 24

2.7.1 Motivations 24

2.7.2 Initial Implementation 24

2.7.3 Limitations of This System 24

2.8 Recent Developments 24

2.8.1 The Tag-Value Parser 24

2.8.2 Flat File Indexing 24

2.8.3 Annotation Types 24

2.8.4 Enhanced Feature Filters 24

2.8.5 Change Hubs 24

2.8.6 Bit Packed Sequences 24

2.9 Conclusions 24

Chapter 3 HMMs for whole *Plasmodium Falciparum* Chromosomes 24

3.1 Introduction 24

3.2 Simple HMM Architectures 24

3.2.1 Methods 24

3.2.2 Results 24

3.3 HMM Architectures with Complementary Emission Distributions 24

3.3.1 Methods............24

3.3.2 Results............24

3.4 First Order HMMs with Time-Reversible Transition Probabilities............24

3.4.1 Methods............24

3.4.2 Results............24

3.5 Discussion............24

3.6 Future Directions............24

Chapter 4 Investigation of Recombination Rates Using SVMs............24

4.1 Introduction............24

4.1.1 Support Vector Machines............24

4.1.2 BioJava APIs for Support Vector Machines............24

4.2 Methods............24

4.2.1 Searching for a Signal Affecting Recombination Rates Using a Word-Frequency Kernel Function............24

4.2.2 Construction and Training of an SVM for Predicting Recombination Rate............24

4.3 Results............24

4.3.1 Recombination Rates Predictions............24

4.3.2 Cross-Validation............24

4.4 Discussion............24

Chapter 5 RVMs for Classification of Expression Data....................24

5.1 Introduction....................24

5.2 Cellular Responses to Doxorubicin....................24

5.3 Generalized Linear Models....................24

5.4 Micro-array Classification Using a Support Vector Machine Implemented as a Linear Kernel RVM....................24

5.4.1 Framework for Generalised-Linear-Models amenable to Expression Arrays....................24

5.4.2 RVM Analysis Using the Small Working Set Heuristic....................24

5.4.3 Function of Genes Identified by GLM Models....................24

5.5 Conclusions, Applications and Future Work....................24

Concluding Remarks....................24

References....................24

Table of Figures

Figure 3-1 Emission probabilities for the four pair-state model 24

Figure 3-2 Diagram of the *P. Falciparum* chromosome 3 and the state paths through three models 24

Figure 3-3 Diagram of the *P. Falciparum* chromosome 2 and the state paths through three models 24

Figure 3-4 Emission Spectrums for all Pair-State Models 24

Figure 3-5 Diagram of the alignments of the 3,4 and 5 state-pair models to Malaria chromosome 3 24

Figure 3-6 Diagram of the alignments of the 3,4 and 5 state-pair models to Malaria chromosome 2 24

Figure 3-7 Counts for Biological Feature and States for the 2-5 Pair-State Models ... 24

Figure 3-8 Normalized Counts of States for Biological Features 24

Figure 3-9 Normalized counts of Biological Features for States 24

Figure 4-1 Comparison of physical and genetic distances along chromosome 22 24

Figure 4-2 Total Results of Training the SVM using Uniform Counts 24

Figure 4-3 Moving Average for Uniform Counts models of Depth 4-6 24

Figure 4-4 Moving Average for Uniform Counts models of Depth 7-9 24

Figure 4-5 Total Results of Training the SVM using Normalized Rates 24

Figure 4-6 Moving Average for Normalized Rates: Depths 4-6 24

Figure 4-7 Moving Average for Normalized Rates: Depths 7-9 24

Figure 4-8 Accuracy for Recombination SVMs Under 3-Way Jack-knifing 24

Figure 4-9 Predictions Across the Entire Chromosome from the 3 Jack-knife Models for Depth of 5 24

Figure 5-1 Scatter Plot of the Two Topoisomerase II Probes Used. 24

Figure 5-2 Expression Levels for Each Probe Used 24

Figure 5-3 Average Weighs Across Relevant Models..24

Figure 5-4 Average Weights Across All Models...24

List of Tables

Table 3-1 Forward-strand and reverse-strand counts....24

Table 3-2 State-transitions and their reverse-complements....24

Table 5-1 GLM for all before-after pairs (to 4 s.f.)....24

Table 5-2 Genes used by cross-validation models....24

Table of Equations

Equation 1-1 A Hypothesis Function 10

Equation 1-2 Error of a Hypothesis 11

Equation 1-3 Some Error Functions 12

Equation 1-4 Dot Products for Items Decomposable into Sub-Spaces with Dot-products Defined 14

Equation 1-5 Definition of Kernel Functions 15

Equation 1-6 A Polynomial From a Two-dimensional Coordinate to a Coordinate Containing One Component for each Possible Product Involving up to Two Dimensions 15

Equation 1-7 Dot products between two polynomial mappings reduced to terms involving the dot product of the unmapped variables 16

Equation 1-8 Polynomial Kernel Function 16

Equation 1-9 Definition of a Probabilistic Hidden Markov Model 21

Equation 1-10 Emission and Transition Probabilities 22

Equation 1-11 Definition of All Legal State-Sequences 23

Equation 1-12 Likelihood of Observing a Given Sequence and Labelling 23

Equation 1-13 Common Dynamic Programming Recursions as Applied to Probabilistic Hidden Markov Models 24

Equation 4-1 Equation of a Plane 24

Equation 4-2 Normal to a Plane as a Weighted Sum of Vectors 24

Equation 4-3 Definition of a Support Vector Machine..24

Equation 4-4 Basis Functions for Kernel Functions and Data Points........................24

Equation 4-5 SVMs in Terms of Basis Functions 24

Equation 4-6 The Normalizing Kernel 24

Equation 4-7 SuffixTree Kernel...24

Equation 5-1 Bayes Theorem...24

Equation 5-2 Rearrangement of Bayes Theorem..24

Equation 5-3 Bayes Theorem in Words..24

Chapter 1 Introduction

The emerging field of Bioinformatics bridges the previously distinct worlds of computer science and biology. Recently, the volumes of information that can be collected with relative ease and moderately low cost per measurement have become vast. With the ever increased the volumes of data, it is no longer possible to analyse all of the data by hand. Computational methods are being developed to generate and test hypothesises and to collate and present these to users. Often, these users are not themselves programmers but biologists. Programs like BLAST (Altschul, Gish et al. 1990) have changed from being of interest to a small group of dedicated programmers to being a tool used daily by researchers in experimental “wet” labs throughout the world.

Established approaches for analysing biological data overlap with methods used in other subject areas. Neural networks have been applied to a variety of problems such as predicting the sub-cellular location of proteins (Reinhardt and Hubbard 1998), splice-site prediction (Rampone 1998) and secondary-structure assignments for proteins (Rost and Sander 1994). Hidden-Markov-Models (used extensively in speech-recognition) have been used as the theoretical basis for a plethora of tasks involving the labelling of DNA or protein sequences. These include gene finding (Burge and Karlin 1997; Birney and Durbin 2000), elucidating evolutionary relationships (Smith and Waterman 1981) and discovering conserved motifs in proteins (Grundy, Bailey et al. 1997). Expression data has been extensively analysed using a wide range of methods. These range from very simple techniques like ranking genes by the difference in absolute level between two conditions (for example, see (Butte, Ye et al. 2001) and references therein) through to more complex methods like

cluster analysis (Eisen, Spellman et al. 1998) and grouping by mutual information (Butte and Kohane 2000). Above all, simple statistical models have been used pervasively for almost all tasks.

With the rapidly increasing size and variety of biological datasets that must been considered in any analysis, there has been a corresponding need for software frameworks to enable the manipulation of these large datasets and aid in their analysis.

1.1 Existing Software Development Frameworks for Bioinformatics

There are a variety of standard activities in bioinformatics that have the potential to be addressed through the use of integrated software packages. These include data visualization and mining, database management, naming and directory services and machine learning. The major advantages of using integrated software packages are that they enable a user to carry out complex tasks without having to re-implement functionality such as file parsing, algorithms and the resource management associated with large datasets. This enables their use by those without the necessary computer skills required to efficiently implement complex or efficient algorithms. The effort involved in developing and maintaining production quality code to address these issues is considerable and usually outweighs the effort required to become familiarised with a package, its interfaces, design and peculiarities. When the package is a community project every user benefits from any user's contribution to and debugging of the code base.

When we started BioJava, there were many bioinformatics-related applications written in almost every conceivable language. Some of these (e.g. HMMER (Eddy 2001) and BLAST (Altschul, Gish et al. 1990)) distribute source code under an open

license. However, usually these applications were coded in isolation from others, so that each time a developer needed a parser for a given file format, or a data structure for some biological entity, they would need to develop their own. There were a handful of toolkits or APIs available under licensing agreements that were compatible with free use by third parties. There were also a few toolkits available commercially, which generally made them difficult to use in an academic setting.

1.1.1 The NCBI Toolkit

The National Centre for Biotechnology Information (NCBI) was founded in 1988 to support bioinformatics in the United States[1]. One of the services it provides is a toolkit written in `C` for the development of bioinformatics applications[2]. The NCBI uses this toolkit internally for managing GENBANK (Benson, Karsch-Mizrachi et al. 2003) and other databases, as well as several applications including BLAST. The current version of the toolkit has data structures for biological sequences, genetic maps, genome assemblies and bibliographical references, as well as many of the other commonly encountered concepts and data-structures in bioinformatics. There is an API for both reading and writing ASN.1[3] documents, and support has recently been added for XML[4] documents. ASN.1 is used as the definition language within the toolkit for data structures. The basic data structures and bookkeeping functions, such

---

[1] See http://www.ncbi.nlm.nih.gov/About/glance/ourmission.html for more information about the NCBI

[2] see http://www.ncbi.nlm.nih.gov/IEB/ToolBox/SDKDOCS/INDEX.HTML for a full listing of the functionality of the tool box

[3] see http://www.asn1.org/ for information about ASN.1

[4] see http://www.w3c.org/XML/ for resources relating to the XML standard

as object life cycle, serialization and de-serialization are generated directly from the ASN.1 definitions, and are therefore named in a consistent manner.

The NCBI toolkit has had fairly limited use as a development platform outside the NCBI. This has probably been because although the source code is available, it has never been regarded as a community project, starting as it did before the emergence of the open source movement. There are also difficulties inherent to developing and maintaining portable C libraries.

1.1.2 Bioperl

Perl[5] is a loosely- and dynamically-typed scripting language that became adopted as the scripting language of choice of bioinformatics during the 1990s. This is due to Perl's ample abilities to act as a scripting language, its powerful regular expression handling and its file manipulation abilities. In 1995, the Bioperl (Stajich, Block et al. 2002) project was formed. From the beginning, it was organized around a web site[6] and there was a strong commitment to open source development and to sharing source code between developers using CVS[7]. It started off as a group of biological scripts, and it quickly became apparent that there were common and reusable concepts used by many different scripts. The first and most important of these was the 'Sequence' object. As of the 1.2 release of Bioperl in 2003, the exact definition of the sequence object is still evolving.

---

[5] The Perl web site can be found at http://www.perl.org/

[6] BioPerl is co-ordinated via the http://www.bioperl.org/ web site

[7] See http://www.cvshome.org/ for more information about CVS

Around 1997, the Bioperl project moved in focus from being a collection of Perl scripts to being a library of Perl modules that defined objects. Soon after that, the project started to adopt the practice of defining abstract classes or interfaces for these data types and then extending these for specific implementations.

Perl in general and Bioperl in particular has since proven to be very effective as a way to glue multiple applications together in pipelines[8]. Large scale systems have been built upon Bioperl, such as the Ensemble genome annotation project (Hubbard, Barker et al. 2002). Bioperl still has resource and computational issues when managing very large numbers of 'live' objects and with allocating and deallocating objects repeatedly. These are mainly due to inherent limitations of how Perl 5 represents objects.

At the time BioJava was started, Bioperl essentially consisted of a module for representing sequences and annotations on those sequences, parsers for a few common sequence formats (EMBL (Stoesser, Baker et al. 2003), SWISS-PROT (Boeckmann, Bairoch et al. 2003), GENBANK (Benson, Karsch-Mizrachi et al. 2003)) and parsers for some commonly used applications (primarily BLAST).

1.1.3 EMBOSS

Up until the mid 1990s, the commercial software package GCG (Womble 2000), written in `C`, was distributed along with its source code. It provided a collection of command-line tools for sequence manipulation. Because the source code was available, many new applications using the GCG libraries were developed and

---

[8] See http://www.biopipe.org/ for more information about BioPipe

distributed in a package called extended-GCG (EGCG[9]). When the license agreement for GCG was changed (around the same time that GCG Ltd was acquired by Oxford Molecular), the source code ceased to be made available. The developers of EGCG started to develop the European Molecular Biology Open Software Suite (EMBOSS) (Rice, Longden et al. 2000). This is a free, open source package containing a wide range of tools for sequence analysis and database access, as well as data-visualisation.

At the core of EMBOSS there is a set of libraries for common tasks, such as sequence input/output (IO), memory management, documentation of source code, and meta-data for command-line parameters. Although most users of EMBOSS are probably not programmers, it does provide a relatively effective library for handling these mundane tasks.

The history of GCG and EMBOSS has underlined the need for widely used libraries to be available to the community that uses them, without fear of their future removal, regardless of how benevolent the current owners may be.

1.2 BioJava

In 1997, Java2 was released, together with version 1.2 of the SDK. This was a substantial improvement over previous versions of Java, both in terms of performance, and in the range of functionality provided by the standard libraries. With this development, it became practical to consider developing a Bioinformatics software package in Java. It was at this point that I first prototyped a set of interfaces in Java which went on to become the core of BioJava. I was familiar with both `C` and Perl, but rejected them for the reasons described below.

[9] The original EGCG web site has been taken over by the EMBOSS site and no longer exists

C, while being a good language for developing high-performance applications, is not always ideal for code reuse and rapid application development. C can be bound to Java applications via the Java Native Interfaces. However, it is easier to manage a project if it is entirely or mainly in one language. Also, the use of native code stops the Java application from being platform-neutral.

Bioinformatics applications often require large and complex data structures. Perl's capability for handling these structures is limited by two main factors. Firstly, it is difficult to handle objects that contain cyclic references, because Perl uses a reference-counting garbage collector that will not remove them, and there is no way to have a non-counted reference. Secondly, allocating many Perl objects is expensive, particularly in terms of the memory foot-print associated with each instance. Many bioinformatics tasks require very large numbers of entities to be compared. Java has a garbage collector that handles arbitrary graphs of objects. Also, the overhead of a Java object is minimal (a couple of words for synchronization and other book-keeping tasks).

At the time, there were no widely used bioinformatics toolkits written in Java. The Neomorphic toolkit[10] was available commercially and provided some visualisation tools that could be embedded within applications. However, it did not provide code for flexible file reading and writing. Also, the underlying model for the sequence was defined in terms of strings and arrays of characters. These do not scale to sequences the size of whole chromosomes.

---

[10] The Neomorphic web site can be found at https://www.Neomorphic.com/das/ngsdk/

It was in this context that BioJava (Pocock, Down et al. 2000) was started, with the aim of providing APIs for common sequence-related bioinformatics tasks for Java applications. The original design was heavily influenced by the Bioperl object model at that time, and since then the two projects have had a degree of common design due to constant comparisons between how each project approaches issues. The core BioJava application programming interfaces (APIs) have been essentially stable since 2001.

BioJava was started in 1999, and became part of the Open Bioinformatics Foundation[11] (OBF) in January 2000. The OBF is an umbrella organisation for the open source Bio* projects. These projects together strive to provide programmer-friendly toolkits in several languages. Currently there are affiliated projects in Perl, Java, Python and Ruby. There are also the CORBA, XML and SQL Bio* projects that are language-neutral but provide data-formats and API interoperability between the language-specific projects.

1.3 Machine Learning

Unlike other bioinformatics toolkits, BioJava was developed from the start to provide a framework suitable for computational biology analysis by machine learning. The main concepts of machine learning are therefore described here together with an outline of how these are supported by BioJava. How these various implementations are used is addressed in Chapters 3, 4 and 5.

The majority of machine learning techniques used in this field can be described as either acting upon discreet entities (by classification or regression) or as labelling a

[11] See http://www.open-bio.org/ for more information about the OBF

sequence of observations (by signal analysis). Machine learning approaches can also be further divided into two main categories: supervised and unsupervised learning. In the case of supervised learning, a training set is available with labelling giving the "true" outcome for each example. For unsupervised learning, the objective is to detect patterns within data for which there is no *a priori* labelling, i.e. to investigate if the data has any inherent interesting structure.

The generalisation of a supervised learning method is how well it treats data that did not form part of its training set. It is desirable for supervised learning methods to generalise well so that the user can have confidence that predictions it generates are trustworthy, even if the new data bears little resemblance to the training data.

A critical consideration in the design of BioJava has been constructing the underlying data structures in such a way that they are appropriate for publishing data to machine learning algorithms. The following sections discuss the way that classification, regression and signal analysis tasks can be represented mathematically. This leads to a natural way for structured biological data to be used in machine learning techniques. While it is not essential to represent the data and interfaces in this way, it does provide us with a common and clear framework upon which we can build. This makes it much easier to change the representations of the underlying data that is exposed to the machine learning technique as well as enabling the evaluating of a range of different machine learning techniques on the same data.

1.3.1 Clustering, Classification and Regression for Single Items

Regression is used to predict a continuous function for data items from a set. For example, regression could be used to predict rainfall levels from measurements of atmospheric conditions. Classification is used to divide a set of items into disjoint

subsets. An example would be the classification of predicted transcripts into the sets of expressed genes and psuedogenes, based on properties of their sequence. Clustering data items produces a hierarchy of relationships, which can be represented as a tree, with data items at the leaves. For example, phylogenetic trees are the result of clustering the sequences of a protein family.

When presented with some items for clustering, classification or regression, it is often natural to think in terms of these items having features, which may be either directly observed (intensity of fluorescence on a micro-array), or calculated (BLAST scores). The analysis is performed on these features. Traditionally, a lot of hard work has gone into defining informative features (for example, different scoring functions for sequence alignments) or for extracting useful information from them (for example, Fourier transforms for expression profiles (Chen, He et al. 1999)). Standard machine learning techniques can be applied to any set of items which can themselves be described by a set of features. We will now consider how these datasets can be represented in a way that makes them applicable to machine learning techniques. This has a direct bearing upon how the interfaces in BioJava have been designed.

Given a set of items X and a set of possible outcomes Y, a space $X \times Y$ of observations and some set of functions called the 'hypothesis space' H, we wish to chose a hypothesis which is a 'good' hypothesis for our data:

Equation 1-1 A Hypothesis Function

$$x \overset{h}{\alpha} y \text{ where } x \in X, y \in Y, h \in H$$

The methods differ in the types of hypothesis spaces that can be searched and the definition of a 'good' hypothesis. In the case of clustering, Y is the space of all

possible trees. The set of trees may be restricted to binary trees, trees that have a depth of 1 (when partitioning into disjoint groups), or may be any other arrangement. For classification Y is the set of labels. For regression, Y is the set of possible values for the continuous variable being predicted: often a range of real numbers. The case of partitioning the data into disjoint groups (classification) is in practice very similar to regression, as the regression case can be thought of as a special case where the output is one of a very large number of possible groups (one per real number).

It is always useful to quantify the error associated with a hypothesis. This can be used during training to help select a hypothesis, and after training to evaluate the success against unseen data. For supervised learning, the error is a measure of how far the predictions fall from the true values. For a wide range of classification and regression tasks, the error of a hypothesis over a complete data set can be treated as the sum of the errors for each individual data point. The exact choice of how to measure this distance between predicted and expected output depends upon the model being considered.

Equation 1-2 Error of a Hypothesis

$$\gamma = \sum_{(x,y)\in X\times Y} err(\delta)$$
$$\delta = |y - h(x)|$$

Where δ is the difference between the expected and predicted value, γ is the total error and err is the error function being used. For unsupervised learning there is no notion of a "true" value, but it is still possible to define a function that is analogous to the error function that is based purely on the assumptions of the model.

The error function is part of how the method will attempt to treat outliers, how sensitive it will be to ‘fuzzy’ data and how general the resulting model will be. Below are some example error functions:

Equation 1-3 Some Error Functions

$$lin(\delta) = \delta$$
$$sqr(\delta) = \delta^2$$
$$hr(\delta, d) = (\delta < d)\begin{cases} 0 \\ \delta - d \end{cases}$$

The third case in Equation 1-3 is interesting, because it includes an insensitive region of width d around the true solution, and any prediction falling in this region receives no penalty. In effect, this is saying that errors up to a value of d are unimportant.

During training, given the training examples T of the form (x, y), the aim will be to select some function that has a low error value. When used for prediction, we will estimate the value of $(y \mid x)$ as $y \approx h(x)$. The ability of the function to generalise can be estimated by computing $h(x)$ for known outcomes that were not part of the training data. A model generalises well if the accuracy of the predictions on unseen data is comparable to the accuracy when predicting outcomes on the training set. This can be estimated by training on a subset of the available training data, and then doing a blind prediction on the rest and calculating the error function or observing the rate of correct and incorrect predictions.

There exists a trivial function that is the exact map defined by T as long as each x appears exactly once (i.e. there is no conflicting information). This function will contain no errors. It has exactly as many parameters as T has members. This has no

generalisation power, as the function is not defined for any x not represented in T. If the training method allows this hypothesis or any similar hypothesis to be chosen then it is 'over-fitting' the training data. If the resulting model contains more free parameters than there are training items (assuming that they really can vary independently), then it is very difficult to ensure that the model is not over-fitting. A good solution to this is to encourage the method to produce models with significantly fewer parameters than there are training examples.

Having defined the problem, we will now discuss a convenient representation of data and hypotheses that enables their easy integration into BioJava.

We can think of the training process as being the selection of the transformation that project objects in X until they superimpose with their image in Y with sufficient accuracy. If, for example, the error function was δ^2 then the problem becomes the estimation of a matrix that performs the rotation of a least-squares fit on the transformed image of X.

In the case where Y has a single dimension, the projection must remove all except one dimension from the feature space. This can be visualised as measuring the distance from points to a hyper-plane (e.g. a linear surface that is one dimension lower than the feature space). This distance is equal to the dot product of the data point with the equation of the plane[12].

[12] For two numbers, $a \cdot b$ is the product of a and b. For vectors, there are two common types of product – the inner and outer products. These can be explicitly disambiguated as $\langle a,b \rangle$ and $a \times b$. Inner products have scalar values. They are often written as $a \cdot b$ and consequently called dot-

There are a range of machine learning methods that can be represented in the form of a sum of dot products. These include classification, regression, k-means clustering and principal component analysis. There are techniques available to adapt this family of methods so that they can be generalised to functions considerably more complex than the linear dot product. This allows these machine learning techniques to consider a much more interesting range of problems. In the rest of this section, we will discuss one method of generalisation; kernel functions.

We can generalise the use of dot products by exploiting the representation of each item as a set of features with associated values. For two data items p and q of compatible types, the natural inner product is the sum of the products of their corresponding features. In the following equation, i is used to index each feature.

Equation 1-4 Dot Products for Items Decomposable into Sub-Spaces with Dot-products Defined

$$p \cdot q = \sum_i p_i \cdot q_i$$

Notice that the dot product requires each feature of the data item to have a dot product defined. For numbers, this is just the normal numerical product. However, the value of a feature may itself be a complex structure composed from a set of features with values.

Dot products have the properties that (a) they are symmetrical functions, (b) that $x \cdot x \geq 0$ for all values and (c) that the value of the dot product is only zero if one or

---

products. In this text, $a \cdot b$ is used where the normal Cartesian dot product is meant, and $\langle a, b \rangle$ is any function that is an inner product of a and b in some space.

both of the arguments has a magnitude of zero. When considering representing features in terms of dot products, it is necessary to ensure that they satisfy these constraints. For example, it would be invalid to use Blast sequence alignment scores as the value for a dot product, as they are not symmetrical.

It is often useful to first transform data from its natural coordinate system (data-space) into one another coordinate system (the feature space) in which particular types of analysis are easier. If ϕ is a function that maps from data-space to feature-space, then the dot product of two items a and b in that space is $\phi(a) \cdot \phi(b)$. If there is a function k such that $k(a,b) = \phi(a) \cdot \phi(b)$, then k is a kernel function. More explicitly:

Equation 1-5 Definition of Kernel Functions

$$k^{\phi}(a,b) = \phi(a) \cdot \phi(b)$$

An interesting subset of kernel functions are equivalent to functions of the data-space dot product. For example, given two vectors, we could define a transform that projected the items into the space of all possible polynomial interactions of order 2 or less (Equation 1-6) which allows conics to be constructed in the data-space. This can be expressed in terms of dot products in the data-space (Equation 1-7). This form can be generalized for data-spaces with any number of dimensions, and for polynomial interactions of any order (Equation 1-8).

Equation 1-6 A Polynomial From a Two-dimensional Coordinate to a Coordinate Containing One Component for each Possible Product Involving up to Two Dimensions

$$P : x \; \alpha \; (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1{}^2, x_2{}^2)$$

Equation 1-7 Dot products between two polynomial mappings reduced to terms involving the dot product of the unmapped variables

$$\begin{aligned} P(a)\cdot P(b) &= (1,\sqrt{2}a_1,\sqrt{2}a_2,\sqrt{2}a_1a_2,a_1{}^2,a_2{}^2)\cdot(1,\sqrt{2}b_1,\sqrt{2}b_2,\sqrt{2}b_1b_2,b_1{}^2,a_2{}^2) \\ &= 1+2a_1b_1+2a_2b_2+2a_1a_2b_1b_2+a_1{}^2b_1{}^2+a_2{}^2a_2{}^2 \\ &= (a\cdot b)^2+2(a\cdot b)+1 \\ &= (a\cdot b+1)^2 \end{aligned}$$

Equation 1-8 Polynomial Kernel Function

$$k^{poly(n)}(a,b)=(a\cdot b+1)^n$$

In terms of the time-and-space constraints, this reduces the problem of finding all polynomial interactions between all elements of two vectors from having complexity that scales badly on the length of the vectors to one that scales linearly. Explicitly computing polynomial interactions of order 5 would require time and space proportional to the fifth power of the length of the vectors, where as using an appropriate kernel function the cost would still be linear.

The feature spaces for many kernel functions are very large compared to the data-space. For almost any training set, the feature space will have more dimensions than training examples, and may be too large to represent explicitly. However, if the dot products can be calculated as a simple function of the data-space dot product, then the feature space size is no longer a constraint to calculations.

The machine learning methods that can be represented in terms of sums of dot products can often be adapted to work with kernel functions, allowing them to explore solutions in the feature space of the kernel, while maintaining performance characteristics related to the dimensionality of the data-space.

It is possible to compose new kernel functions from other kernel functions and scalar functions. In all cases, care must be taken to not invalidate the three properties of dot products defined above. Here are three examples:

1. $af + bg$ Equivalent to concatenating the feature spaces of the kernels f and g after scaling them by a and b respectively.
2. $f \cdot g$
3. $\xi(f(a,b))$ where ξ is any scalar function that maintains the conditions that apply to dot-products, such as the polynomial kernel in Equation 1-8.

The BioJava support vector machine implementation (discussed in 4.1.1 and 4.1.2) provides a linear dot product kernel implementation for sparse vectors of real numbers. Additionally, there are a range of kernel functions that are implemented in the third form above that return some function of the result of another kernel function. These include kernels for radial basis functions, polynomials and hyperbolic tan. There are also kernels that implement normalising transformations, such as projection onto a unit sphere. Data is made available by implementing the kernel function interface so that it returns a dot product for some pair of Java data structures. Then, an appropriate feature space can be constructed by composing the kernel function objects as required. This affords a great deal of flexibility in the range of feature spaces that can be explored.

SVMs, described in Section 4.1.1, are one of a family of models called Generalised Linear Models. Another related form of linear model is trained using the Relevance Vector Machine methodology, described in 5.3. Applications of the representations described here will be discussed in Chapters 4 and 5 with examples.

1.3.2 Signal Analysis with Hidden Markov Models

Signal analysis methods deal with data that are composed from a linear sequence of observations, possibly of differing lengths. One approach to signal analysis used widely in bioinformatics is to infer properties of the structure of the sequence based upon a model of how the sequence may have been generated. The sequence can be represented as a series of observations x which are indexed in the form x_i where if $i < j$ then x_i is before x_j in the sequence.

Probabilistic Hidden Markov Models (HMMs) (Durbin 1998) are generative models that have been applied to a wide range of biological problems since their introduction to computational biology (Churchill 1989; Krogh, Brown et al. 1994). Formally, they define a probability distribution over all sequences that can be generated using the production rules of a stochastic regular grammar. One benefit in representing models as HMMs over stochastic regular grammars is that HMMs can be easily visualised as graphs, whereas stochastic regular grammars are inherently textual.

A common and successful application of HMMs is the modelling of a family of evolutionarily related sequences. A popular form of model for this kind of application is the profile HMM, where a sequence of match states through the model represents the consensus sequence for some biological feature. Insertion and deletion states model the corresponding evolutionary events. Profile HMMs form the basis of the SAM package (Hughey and Krogh 1995), and profiles built with the HMMER package (Eddy 2001) form the basis of the Pfam database (Bateman, Birney et al. 2000).

Although profile HMMs are a widely used form of HMM in computational biology, it is possible to build much more flexible models. For example, Meta-MEME

(Grundy, Bailey et al. 1997) takes simple ungapped weight-matrices, which are based on motifs discovered using the MEME package (Bailey and Elkan 1994), and links these together with spacers to form higher-order models.

Another common type of HMM is an alignment, or pair HMM. This form emits correlated pairs of sequences. Pairwise alignment algorithms can be represented in this form. The Dynamite package (Birney and Durbin 1997) provides a language for implementing new pair-HMM algorithms. However, Dynamite itself does not provide any facilities for training the parameters for these models. This means that Dynamite models must be parameterised by hand, a process which is more of an art than a science.

HMMs can be trained from labelled training data. That is, given a set of sequences where the model states have been assigned, the optimal probabilities for the model can be calculated directly. The observation counts are normally regularized using an appropriate background model that reduces the possibility of over-fitting the examples. It is generally accepted that regularization enhances the generality of the model to unseen sequences.

The most commonly used forms of regularization are Dirichlet priors (which are equivalent to pseudocounts) and Dirichlet mixtures (Brown, Hughey et al. 1993; Sjolander, Karplus et al. 1996). Dirichlet priors represent probability distributions over the range of possible counts, and are used to blend the probability obtained using the raw counts with the expected counts if the null model were true. This is implemented by adding extra “pseudocounts” to the observed counts. Dirichlet mixtures work in a similar way to Dirichlet priors, but in this case there are multiple

prior models, and the prior model which is closest to the observation has the most weight during the blending.

It is possible to use other priors, such as multinomial Gaussian distributions and their mixtures over log-odds space (O'Hagan 1994), but these have not been extensively investigated for biological models. This is probably because as Dirichlet priors can be relatively easily implemented and have been applied successfully, there is no perceived reason to use different types of prior models.

In contrast to training from fully labelled data, HMM parameters can be estimated from an unlabeled set of sequences given a model architecture. This is achieved iteratively by estimating a probability distribution over all possible labellings for each sequence given a current set of estimated model parameters. Counts can be added in proportion to the probabilities of the labellings, or sampled from this distribution. The counts are then normalized and regularized, and these new parameters are used as the starting point for the next round of parameter estimation. This cycle is repeated until the model parameters cease to change by any significant amount or a pre-determined number of cycles have elapsed. It is usually sufficient to start with arbitrary random parameters, given a model with few enough free parameters. When counts are added in proportion to the probability distribution over all possible labellings, this procedure is known as Baum-Welch training (Baum, Petrie et al. 1970; Rabiner 1989; Durbin 1998). When adding counts by sampling from this distribution, we have called this procedure Baum-Welch with sampling.

Finally, complex models can be parameterised using a mixture of labelled and unlabelled data. For example, models for distinguishing between protein secondary structure elements may have complex models for α-helix and β-sheet involving

multiple repeating patterns of states. Training data may be binned into sets for both secondary structure elements, and then maximum-likelihood used within the bins (Asai, Hayamizu et al. 1993). This initial splitting of the data is equivalent to a partial labelling of the data that restricts those observations to being generated by a sub-set of the parameters.

HMMs can be applied to a wide range of sequence analysis tasks. The BioJava dynamic programming toolkit was intended to allow the implementation of a wide range of these algorithms through a consistent API. To achieve this, it was helpful to find a very general description of HMMs and the algorithms that are used to manipulate them. A good formal representation of this general description aids in developing clear APIs and good procedural implementations of the procedures described above.

Equation 1-9 Definition of a Probabilistic Hidden Markov Model

$$M = (\Omega, I, \Sigma, e, t); \Sigma = I^2$$
$$M = \text{model}; \Omega = \text{emission alphabet}; I = \text{states}; \Sigma = \text{transitions};$$
$$e = \text{emission probabilities}; t = \text{transition probabilities}$$

We can represent an HMM as a tuple of parameters (Equation 1-9). The model emits symbols from an alphabet (Ω), such as DNA. It has a finite set of states (I), often called the state-space. The model has a set of transitions (Σ) defined as all ordered pairs of states. There is a probability distribution over the transitions (e) and another over the members of the alphabet emitted by each state (t). It is often convenient to represent the emission and transition probabilities as being an array of functions dependant upon the current state under consideration (for example,

Equation 1-10). In an object-oriented interpretation, these functions could be modelled as being properties of a state.

Equation 1-10 Emission and Transition Probabilities

$$
\begin{aligned}
& k \in I \\
& e_k(a) = p(a \in \Omega \mid k) \\
& t_k(j) = p(j \mid k) \forall (k, j) \in \Sigma
\end{aligned}
$$

It is common for some values of $t_k(j)$ to be constrained to always be zero. In this case, we can consider there to be no legal transition from state k to state j. We can describe these states as being unconnected. A small extension to the original model redefines the transition term as $\Sigma \subset I^2$. From the point of view of implementing efficient algorithms that act upon these finite state machines and of efficient data structures for storing parameters, it is often important to know which states are explicitly unconnected, rather than happening to have a transition probability set to zero by a particular parameterisation.

For example, in an HMM that models a weight-matrix of length 100, there are 100 states, one for each column of the weight-matrix. The complete transition matrix would contain 10,000 elements. However, in the HMM for a weight-matrix, the only state that can be reached from a given state is the single one that represents the next column. The HMM representing the weight matrix would have 99 legal transitions in total. It would be an inefficient use of resources to store the square transition matrix when it is only necessary to use an array linear on length to the number of states in this model.

A sequence can be labelled with states so that there is one state associated with each observation, and each transition represented by neighbouring pair of states are legal in

the HMM. The sequence of states is called a state-path. Formally, this can be written as:

Equation 1-11 Definition of All Legal State-Sequences

For each x_i there is a value of $y_i \in I$ such that $(y_{i-1}, y_i) \in \Sigma$

Given both x and y the joint probability of a sequence and its state-path pair can be calculated as:

Equation 1-12 Likelihood of Observing a Given Sequence and Labelling

$$p(x, y \mid M) = \prod_i p(x_i \mid y_i) p(y_i \mid y_{i-1})$$

Given Equation 1-12, it is possible to evaluate any state-path. With a given set of parameters, there will be a set of paths (often just one) that have a higher value than any others do. The Viterbi algorithm (Rabiner 1989; Durbin 1998) finds one of these paths. By summing over every possible state-path, that could have produced a sequence, it is possible to calculate $p(x \mid m)$. The forwards and backwards recursions (Rabiner 1989; Durbin 1998) calculate this value, initialising from the first and last symbol of x respectively (Equation 1-13). In these recursions, it is necessary to loop over the variable indexing x according to its natural ordering (and in the reverse of this for the backwards algorithm), and similarly the destination states for each transition must be looped over such that the recursion has been calculated for every value of the recursion that this step relies upon.

Equation 1-13 Common Dynamic Programming Recursions as Applied to Probabilistic Hidden Markov Models

$$V(\bar{i}, j \in I) = e_j(x_{\bar{i}}) \cdot \max_{k \ni (k,j) \in \Sigma} \left(t_k(j) \cdot V(\bar{i} - adv(j), k) \right)$$

$$Bp(\bar{i}, j \in I) = \arg\max_{k \ni (k,j) \in \Sigma} \left(t_k(j) \cdot Bp(\bar{i} - adv(j), k) \right)$$

$$F(\bar{i}, j \in I) = e_j(x_{\bar{i}}) \cdot \sum_{k \ni (k,j) \in \Sigma} t_k(j) \cdot F(\bar{i} - adv(j), k)$$

$$B(\bar{i}, j \in I) = \sum_{k \ni (j,k) \in \Sigma} t_j(k) \cdot e_k(x_{\bar{i}}) \cdot B(\bar{i} + adv(k), k)$$

$\bar{i}$ is the vector of indecies into the sequences being aligned
$V(\bar{i}, j \in I)$ is the viterbi score for state j at sequence index $\bar{i}$
$Bp(\bar{i}, j \in I)$ is the backpointer from the current state to the previous one
$F(\bar{i}, j \in I)$ is the forwards score
$B(\bar{i}, j \in I)$ is the backwards score
$e_j(x_{\bar{i}})$ is the emission probability of the symbol at that index conditional upon the state
I is the set of states
t_k(j) is the transition probability from state k to j
adv(j) is the vector to add to $\bar{i}$ to represent which directions have been advanced by the state j

For the case of aligning multiple sequences, the observation x can be replaced by the product of all sequences being aligned such that for sequences a and b we end up aligning $x = a \times b$ to the model where $x_{i,j} = (a_i, b_j)$. We can generalise this to any number of sequences being simultaneously aligned, and replace the compound subscript with a vector $\bar{i}$. For the Viterbi, forward and backward algorithms, we can proceed exactly as before, as long as we use the partial ordering of $\bar{i}$ such that adding one to any component of $\bar{i}$ would produce a new vector that comes after it. So that some states can advance through a sub-set of the sequences being aligned (for example, insertion or deletion states in a pair-wise alignment HMM), it is convenient to have an advance vector associated with each state. This is of the same

dimensionality as $\bar{i}$. The advance vector is purely a function of a single state. A valid object-oriented interpretation is for states to expose an advance array as a property.

The BioJava library allows HMMs with arbitrary architectures to be constructed. The HMM APIs are strongly modelled on the definitions of HMMs described above. In particular, the APIs do not directly distinguish between models that generate one, two or any other number of sequences. The sequence of observations presented to an HMM is represented using the BioJava Alphabet, Symbol and SymbolList APIs (2.4), which allows the algorithms to be applied to a much wider range of data than either DNA or character strings without needing to alter the code implementing the recursions themselves, or the associated data models. From the users' point of view, there is no programmatic difference between models that are fully connected and ones that are extremely sparse. Section 2.6 discusses the BioJava HMM APIs. Chapter 3 explores the use of HMMs for modelling chromosomal structure.

1.4 Implementation and Use of BioJava

In the following chapters the implementation of BioJava and its application to real problems are discussed. In Chapter 2, the implementation of the core of BioJava is described. In Chapters 3, 4 and 5, BioJava is applied to particular classes of machine learning problem. HMMs are used to discover data-compressions for whole chromosomes in Chapter 3, SVMs are applied to recombination rate prediction in Chapter 4 and RVMs are used to classify expression data in Chapter 5.

Chapter 2 The BioJava Core Interfaces

BioJava is intended to provide Java based APIs for common bioinformatics tasks. It also strives to be a convenient basis for writing potentially computationally expensive algorithms. To reduce the learning curve, and to decrease maintenance overhead, individual APIs must be complete enough to allow them to be used algorithmically, but slim enough that they can be easily implemented and used. The balance between making the APIs not only powerful but also small, is sometimes difficult to maintain, but has, in my view, fostered a high degree of elegance in the underlying object design.

There are two clearly different cases where code reuse is beneficial. The first, and most commonly thought of case is the reuse of library code by invoking it from multiple applications. For example, it is very common to reuse a matrix mathematics library during numeric programming. We could call this the “new using old” case. The other reuse case is when library code can execute a tried-and-tested procedure that in turn calls some application specific code. For example, in Java, a listener can be registered with a window to handle mouse movement events. In this case, the library code is responsible for drawing the window and for invoking the listener of mouse movements, but the exact behaviour of the library is customized by the listener. We could call this the “old using new” case.

The design and implementation of the BioJava libraries has primarily been an exercise in computer science, not biology. Throughout, we have striven to foster a high degree of code reuse both by providing APIs that can be used in a wide range of contexts (new using old), and by providing opportunities for developers to drop in new implementations of these APIs without affecting existing code (old using new).

The APIs relevant to this dissertation, and for which I have been primary designer and implementer, are the following:

- Nested Exceptions and Assertions
- Changeability
- Symbols, Alphabets and SymbolList
- Sequence, Feature and SequenceDB
- Distribution
- MarkovModel
- Query

Wherever possible, the API is defined in terms of Java interface definitions, allowing for the seamless integration of multiple implementations. Indeed, it has been the reliance on interfaces that has made the development of the BioJava library relatively rapid and robust. We have discovered that nearly all core data types can be implemented in multiple ways depending upon a host of factors, so the entire toolkit makes as few assumptions about implementation as are possible.

2.1 Java as a Language for Bioinformatics

Java (Gosling, Joy et al. 2000) is a language created by Sun Microsystems, originally for use in imbedded systems such as mobile phones, watches and ABS systems. It relies upon a definition of a virtual machine (VM) (Lindholm and Yellin 1999) that is responsible for thread and memory allocation, byte-code instruction execution and enforcing security restrictions. The byte-code is naturally object-

oriented and has support for advanced features such as exception handling and thread synchronization. The byte-code acts upon a stack of working variables, an arbitrarily large set of virtual registers and the object (or class) that is currently in scope. There is no pointer type in Java, or in the byte-code, making it impossible to write byte-code that addresses arbitrary memory. Theorem provers can be used to validate that a given portion of byte-code is safe to execute, avoiding some of the issues with other languages (such as invalid memory allocation, executing instructions on inappropriate types and validation that the execution stack is always in a consistent state).

The Java VM is responsible for interpreting the byte-code and for environment functions such as memory allocation and garbage collection (freeing objects from the memory pool once they are no longer within scope), system calls (IO, process execution, thread management) and managing peers to the native operating system graphical user interface (GUI). With a VM of a given version (e.g. 1.2.2) and any platform (e.g. Sun for Windows, Compaq for Tru64), executing a portion of byte-code should produce exactly the same results, even if the performance differs[13]. This code portability by design is historically one of the major benefits of Java. In practice, platform incompatibilities nearly always arise from platform-specific portions of the VM, such as graphical peers, rather than bugs in the execution of byte-code.

---

[13] Since version 1.4 of Java, some floating-point mathematical operations may be implemented by processor-specific instructions that do not conform to the IEEE floating-point maths required by the Java virtual machine specification. In the vast majority of cases, this does not change the result of a calculation greatly enough to alter program behaviour. The keyword `strictfp` can be applied to any class or method that must uses the IEEE-compliant math operations, such as numerically intensive code that needs particular overflow/underflow and rounding semantics.

Pure Java byte-code interpretation has historically been slow relative to native code (compiled from C/C++ or FORTRAN, for example), but has always compared favourably to other interpreted languages such as Perl. Recently, with the move of Java from toy examples and small graphical applications to large, demanding applications such as web-server back ends (especially J2EE[14]) and large-scale numerical processing (for example, the Colt matrix mathematics library[15]), a number of technologies have appeared to improve the performance.

Initially, just-in-time (JIT[16]) compilers increased performance of large blocks of numerical code to that comparable with C++ by compiling each byte-code function into native code for the physical processor once a class was first loaded (present even in many Java1.1 VMs). Some Java byte-code instructions could be represented cleanly as one or more simple native instructions (for example, the arithmetic operations). However, many Java byte-code instructions have no direct representation (such as object allocation, or method invocation), so must be converted into calls to the virtual machine. JIT compilers tend to do a good job of increasing the performance of numeric code that resembled more classically procedural programming styles. JITs proved insufficient for many tasks as many Java methods are small, and very often are executed as virtual calls which can not be resolved at compile-time. In addition, due to the highly polymorphic nature of much Java code, it is often impossible to perform optimisations because the simple type-based system

---

[14] See http://java.sun.com/j2ee/ for information related to the J2EE standard

[15] Colt is distributed from http://tilde-hoschek.home.cern.ch/ ~hoschek/colt/index.htm

[16] See http://java.sun.com/docs/jit_interface.html for more information about Sun's JIT compiler

can't give enough information to know the context within which code will be executed.

The latest family of virtual machines are based upon Sun's Hotspot Virtual Machine architecture[17]. This uses a mixture of several techniques to remove performance bottlenecks and optimise code execution. Firstly, a large portion of Java execution time can be spent in object allocation and garbage collection. This is especially expensive for objects that are allocated and then discarded within inner loops. Hotspot initially flags objects with a creation time, and places them into a nursery area. When more memory is needed, Hotspot first attempts to free objects within the nursery rather than completing a full garbage-collection cycle.

As an anecdotal example of how this can impact performance, I wrote some code that unnecessarily allocated a large number of objects within a tight loop, and then ran the application on a PIII 800MHz with the Hotspot Virtual Machine and a Compaq DS40 with Compaq 1.2.2 Fast VM. After one and a half days, the process on the DS40 had still failed to complete. On the PC, it completed after 110 seconds. Once the unnecessary objects were not being created, the Compaq server took just 20 seconds to execute the code, and the PC took 56 seconds. This clearly indicates the affect of the VM implementation upon performance.

After careful memory management, the second truism of code optimisation is that 5% of the code will account for 95% of the execution time, so if you wish to focus efforts upon optimisation, this is the portion to target. The hotspot VM continually

---

[17] See http://java.sun.com/products/hotspot/docs/whitepaper/ Java_HotSpot_WP_Final_4_30_01.html for more information about Sun's hotspot VMs

profiles the application, and concurrently optimises each of the execution hot spots. This results in applications increasing in performance as they are run (often by factors of greater than 10 times).

Method invocations make many optimisations impossible, especially if the method is bound at execution time rather than at compile time (for example, virtual method invocations). This is because the optimiser does not know what the side-effects of the invoked code will be, so it can't really perform aggressive optimisations to eliminate redundant code or to reorder instructions across function calls.

In traditional languages, this has been tacked with tactics such as macro-expansion, inlining and by defining many methods as not being over-ridden by subclasses, making the linkage static. The hotspot VM takes another approach by dynamically inlining functions to produce multiple context-dependant compiled and optimised versions of a given portion of an application. Most small functions, such as get/set pairs can be trivially inlined, removing the method invocation overhead completely. Increasingly complex methods may be inlined, allowing loop variables to be merged and larger blocks of code to be optimised. Certain types of objects can be proven to decompose to the set of their fields and methods only (i.e. their object reference is never explicitly used to test for identity), in which case the fields can be allocated on the stack. This is similar in spirit to having the power of a templated method that can be parameterised with the template type during run-time. The result of these optimisations is that hotspot-interpreted Java code that is polymorphic or uses many small methods can often execute at speeds comparable to or faster than similarly polymorphic `C++` code. Currently, dedicated procedural style `C` or `C++` may out-perform similar Java code, but even here Java is making inroads. For example, the

Colt matrix maths library in Java now has comparable performance to the FORTRAN matrix math libraries. There is undoubtedly more work to be done for high-end computation in Java, but it is no longer an insurmountable obstacle to the acceptance of Java for the hard end of Bioinformatics.

Bioinformatics is a field that is constantly redefining itself. Some problems are clearly defined, such as the alignment of two proteins using the Smith-Waterman algorithm (Smith and Waterman 1981). However, many other issues are moving targets. There is also the constant pressure to produce results quickly. Traditionally this has caused a polarization between the development of handcrafted applications in languages such as `C` for specific tasks like the BLAST applications (Altschul, Gish et al. 1990), and the use of rapidly developed 'throwaway scripts' in scripting languages such as Perl[18] and Python[19]. In practice, 'throwaway scripts' often become the basis of sequence analysis pipelines that have a lifetime of months or years, and are maintained by a succession of individuals. Eventually, these must be re-written to improve performance, to fix bugs inherent in the initial design, or to allow the application to perform tasks that were not part of the original design aims.

Java is a suitable language for rapidly developing Bioinformatics applications. It can be used to write the computationally expensive as well as the flow-control portions of Bioinformatics scripts. If libraries of biological functionality are developed, and these are easy to use and extend, then it becomes possible to achieve rapid development of throwaway scripts. If these short-term applications become part

---

[18] The Perl web site can be found at http://www.perl.org/

[19] Python is distributed from the http://www.python.org web site

of pipelines, the object-oriented nature of Java code means that it is potentially possible to salvage much of the intellectually expensive code, and to quickly isolate design faults.

The Java compilers are much more pedantic than `C` or `C++` compilers, disallowing many unsafe constructs that can generate strange runtime behaviour. For example, casts are checked where possible, and arbitrary pointers do not exist. Memory allocation and de-allocation are handled by the VM. This means that many errors that would show up as a program crash in other languages cause the Java compiler to generate error messages.

BioJava is intended to provide the functionality needed to rapidly develop effective Java applications for bioinformatics. The design of the language, compiler and virtual machine help greatly in quickly developing robust applications. BioJava builds upon this strong foundation by providing APIs for common biological objects and tasks, such as biological sequences, and reading these from files. Additionally, a number of classes provide basic functionality that increases both the encapsulation and the robustness of BioJava's highly polymorphic code. The rest of this chapter describes the core classes and interfaces that provide this functionality, and for which I was solely or primarily responsible for the design and implementation.

The conventions adopted here for referring to Java types and methods are those used in the Java documentation. When referring directly to types and methods, the type used is `fixed width`. When methods are referred to, the usual form will be to name the method followed by ellipses as in `someMethod()`, regardless of the actual arguments accepted by the method. If it is necessary to describe the parameters accepted by a method, either for the clarity of the text, or to disambiguate over-loaded

methods, the types of the arguments will be included as in `anotherMethod(String, int)`. In a few very rare cases, the names of the arguments must be included as in `substring(int start, int length)` so as to make the semantics more clear. In general, once a method is introduced, it will be referred to using the shortest unambiguous form.

2.2 Nested Exceptions and Assertions

Both during the development of applications and their deployment, failures occur. These may be due to the application being implemented incorrectly, being used with data that it was never designed to be used with, or by some external failure, such as a break in network communication. Programmatically handling failures gracefully and informatively is a key to developing robust software rapidly.

Java supports the handling of error conditions by the throwing of exceptions. The built in exceptions have a constructor that takes a message `String` only. Java methods can be defined as throwing a list of `Exception` types. This means that the method can raise any one of these exceptions if it is unable to complete processing, and that it is limited to this list of checked exceptions. Some exceptions, such as `OutOfMemoryException` are unchecked as it would be difficult to guard against the many places where they may be raised without bloating both the volume of source code and impacting upon run-time performance. During invocation, a method may choose to not raise any of the listed exceptions (indicating that it was successful). Function calls in `C` generally return an error code to indicate error status. In Java, the method would return a value if it was successful or throw an exception if it was unable to complete. The basic exceptions are applicable to the case where the error is

caused by a failure in the program that is clearly attributable to a single action, such as accessing a file, or an array index being out of bounds.

When complex applications are composed of multiple 'black box' modules, failures in one module may cause failures in another module. With classical exceptions, the original cause would either percolate up by allowing the `Exception` to be thrown from all methods in the first module that invoke methods in the second one, or would have to be caught, and a new `Exception` thrown to describe the failure. The first alternative tends to lead to methods throwing very large numbers of specific exceptions defined in other modules or it leads to methods throwing extremely general exceptions that provide poor programmatic control over error handling. This problem becomes even more pronounced when there are multiple implementations of a given interface. The interface author can not possibly foresee all of the ways the interface may be implemented or the range of potential failures, so can not declare all of the exceptions that may be raised by all implementations. For example, if an interface is implemented using a Common Object Request Broker Architecture[20] (CORBA) peer in one case and file access in another, the implementations may fail due to CORBA-specific exceptions or problems with file access, but the original interface author could not have known this, so would not have listed exceptions specific to these two failures in the methods.

BioJava provides subclasses of `Exception` and `Error` (the base-class for unchecked exceptions) that have an extra field that contains a reference to a causal exception. These are called `NestedException` and `NestedError` respectively. In the above

[20] http://www.omg.org/ is the web-site for the organisation that manages the CORBA standards

example, the interface author would declare the methods as throwing sub-classes of `NestedException` describing the type of the failure (something like `RetrievalFailedException`). The CORBA implementation would catch the CORBA-specific exceptions and then construct and throw a new `RetrievalFailedException` instance that refers to the CORBA exception. Similarly, the file-based implementation would catch `IOException` instances and throw new `RetrievalFailedException` instances that refer to the `IOException` that caused the failure. `NestedException` and `NestedError` can be nested to arbitrary depths, allowing a complete 'chain of evidence' to be collected about the cause of errors without requiring modules to have knowledge of all of the exceptions raised by indirect dependencies.

This ability to have both a complete chain of evidence for any failure while respecting encapsulation has made it much easier to develop portions of the BioJava library independently of one another while still allowing them to rely on functionality provided by other BioJava modules. In our view, this has strongly contributed to the rapid development of the libraries.

Because of a combination of Java not supporting sub-classing by restriction, and the Java compiler being pedantic about ensuring that each error condition is accounted for in the code, exceptions must be caught even when they are logically impossible to generate. For example, if counts are being collected for a probability distribution over the DNA alphabet, and the sequence is known to be DNA, then it should be impossible to raise an `IllegalSymbolException`. However, because the component-based APIs don't have this information, the compiler will expect the exception to be handled. The recommended way to handle this is to catch the exception and throw a

`NestedError` instance indicating that an assertion has been violated. The `NestedError` instance will cause the stack to unwind until it is explicitly caught. If not caught, the thread will exit with an error message. In theory, this case should never happen, but in practice, incorrectly implemented objects manage to invalidate these checks, particularly during development, and the assertion failures clearly pinpoint the source of the errors.

Nested errors and exceptions are used throughout the BioJava libraries and in many of the applications that use these libraries. They have proven to be invaluable in writing robust code. Sun has added the concept of nested exceptions to the latest version of Java (1.4), and we look forward to merging our system with theirs.

2.3 Changeability

When developing complex applications, and in particular those which may contain multiple threads of execution, it is important to control which resources may change and which can not. Often, it is also very important to be informed if something does change so that some action can be taken. Unfortunately, the exact details may need to be decided at run-time, so can not be implemented at compile-time as a mutable or immutable interface implementation or by using keywords (such as `C`'s `const` modifiers). Robust applications that are built in a modular manner need strong guarantees about which resources will and won't be modified, and expect these to be enforced.

An object may change state because a method is invoked that would directly modify it. Alternatively, it may change state because some other object, that it delegates state maintenance to, is modified. For example, a `List` instance may be modified by invoking the add method to append an item to the list. A view on the list returned by

`Collections.unmodifiableList(List)` cannot be modified directly, but if the underlying list is altered, then the unmodifiable view will reflect this change. This illustrates the difference between modifiability and changeability. The unmodifiable list is changeable, as there is a legal way for its data to be altered, even though it can't be modified directly.

BioJava contains a complete object model for tracking the changes made to objects, and for allowing changes to be prevented, without breaking object encapsulation. The `Changeable` interface defines methods to add and remove listeners which will be informed when the state of an object alters. These listeners are informed before the object attempts to change, and have an option to veto the change by throwing a `ChangeVetoException`. If none of the listeners throws an exception, then the object updates its state and then informs each listener of the change. At this stage, each listener can synchronize state to ensure data-integrity. In principal, this is a very similar design pattern used by Java Beans to implement 'bound properties', but offers several advantages akin to those provided by a simple cascading transaction processing framework.

Each BioJava interface that extends the `Changeable` interface has public final static fields that hold `ChangeType` instances that encapsulate one way that the Interface implementations may change state. For example, `FiniteAlphabet` defines a static field called SYMBOLS that represents a modification that changes which Symbol instances are contained within the alphabet. Each implementation of `FiniteAlphabet` is required by the Changeability API to allow `ChangeListener` instances to be registered. The `ChangeType` class supports the idea of a hierarchy of change types.

The root of this hierarchy is UNKNOWN. When listeners are added to a changeable, they will be informed of all events that descend from that type.

It is required by the contract described in the Changeable documentation to inform all listeners registered for a given type and all of its descendents whenever state changes by invoking the `preChange()` method on each listener before it changes state If any one of the listeners vetoes the change, or if for any other reason the state cannot be updated, then an exception will be raised. If the change can go ahead, it will commit the state change and then invoke the `postChange()` method on listeners to inform them that the state has been successfully modified. There is no guarantee made about the relative orders that different listeners will be informed either before or after a change is made. It is legal to inform different listeners in different threads, and in some situations this may be a sensible thing to do, if for instance, the listeners need to communicate with network resources.

The `preChange()` method indicates that a change should not be made by raising a `ChangeVetoException`. `ChangeVetoException` extends `NestedException`, allowing a change to be prevented because of a failure elsewhere in the application to publish this information. A common cause for this is when an object stores its state in a delegate. If a method is invoked on the object that would modify the delegate, and the change on the delegate is vetoed then the delegate will raise a `ChangeVetoException`. This will be caught by the object and a new `ChangeVetoException` will be thrown indicating that the requested modification could not be made. This new exception will nest the original exception, allowing the complete reason for the failure to be maintained. One very useful `ChangeListener` is the AlwaysVeto instance. This always throws a `ChangeVetoException` in

`preChange()`, ensuring that the change cannot go ahead. In this way, a mutable implementation of an interface can be instantiated, populated and then locked. From that moment on, no more modifications can be made to it.

The way that the Changeability API can be used to dynamically constrain what data can change, and what data is fixed, can be illustrated by the dynamic programming code. While an HMM is being used for an alignment, `AlwaysVeto` is registered as a listener to all of its parameters, preventing them from changing. Once the alignment has been completed, the `AlwaysVeto` listener is removed, allowing it to be modified again. This kind of fine-grained and dynamic control of which properties can be modified is key to developing robust and modular BioJava functionality.

`ChangeEvent` contains a field to store another `ChangeEvent`. This is used in the case when a change in one object leads to a change in another. For example, if a probability distribution that encapsulates some transition probabilities in an HMM changes then the HMM will no longer have the same parameters. The HMM will listen to each distribution, waiting for them to change. Whenever they do, it will inform listeners that the model parameters have changed with a `ChangeEvent` instance that refers back to the event fired by the probability distribution. Again, by maintaining references back to the event that caused the new event to be fired, a complete ‘chain of evidence’ can be built up about why an object wishes to change, without invalidating the objects encapsulation. This potentially allows listeners to accept or reject a change according to one of the underlying causes.

To avoid the expense of maintaining the entire changeability infrastructure all of the time, several BioJava objects only build the support objects once listeners have been registered. In this way, the cost of having changeability support in an object with no

listeners is the cost of one field in the object that has a null value. Once listeners are added, this field would be filled with a reference to the support data-structures. Mutator methods can easily chose not to perform expensive operations while there are no listeners, such as protecting the listener list with a synchronized block. In addition, if some listeners have been registered, but none have been added that need to be informed of changes to state-delegation objects, then there is no need to instantiate the change-forwarding apparatus. By implementing the classes that will take part in a network of Changeable objects carefully as described here, it is possible to avoid almost all unnecessary overhead.

The Changeability API is used extensively throughout the BioJava libraries. It has proven to be effective at guarding against design flaws and has prevented countless bugs that could have been caused by assuming the involatility of data. It has also been leveraged within the DAS[21] (Dowell, Jokerst et al. 2001) and GUI packages to implement efficient data-caching schemes. In the future, it may be necessary to implement a full transaction-processing framework. Until then, the Changeability API will continue to be an invaluable tool.

2.4 Symbols, Alphabets and SymbolList

Although BioJava must deal with DNA and protein sequences, the underlying interfaces for defining sequences is extremely flexible, allowing almost any signal to be represented as a stream of symbols. This allows all code defined in terms of these APIs to be applied to a wide range of use-cases. For example, the FASTA file format object can be used to read and write state labels from HMMs without any change,

---

[21] The DAS standard, and associated information is published at the http://biodas.org/ web site

simply by parameterising it with the alphabet of states for that HMM, and the sequence viewing APIs can be used 'out of the box' with this data. This section describes these APIs and how they can be used to represent sequences of complex data structures in addition to DNA.

The interfaces borrow heavily from the concepts of an entity, a set and a string. A set is an item that contains some number (possibly zero) of entities. A string is an item that can be represented as an ordered list of entities (possibly zero in length). If all of the entities in a string belong to a particular set, then it is described as a string over that set. For Java `String` objects, the entities are `char` instances, the set is the Unicode Character Set and the string is the `String` class. In BioJava, the `Symbol` interface represents an individual entity, `Alphabet` represents a set of `Symbols` and a `SymbolList` represents a string over an `Alphabet`. These interfaces are designed to be as mathematically elegant as possible, as over time this has made them very useful for seamlessly implementing algorithms. This has had the unfortunate side effect that a large amount of documentation is needed to describe what all of the API does. Fortunately, users of these APIs usually do not need to know the finer details to perform all common tasks.

To reduce the amount of special-case code required, the Symbol interface represents ambiguity symbols, such as 'n' or 'x' and gaps '-' as well as concrete symbols such as the nucleotides 'a', 'g', 'c' and 't'. Sometimes the natural alphabet to work in can be represented as the cross product of other alphabets. When writing code to translate a region of RNA, it is convenient for the code to work with RNA triplets. The natural alphabet for this is RNAxRNAxRNA, which contains symbols that

represent entities like [a, u, g] and [c, c, c]. To allow all of this to be represented consistently, three symbol interfaces extend one another.

`Symbol` is the most generic interface. `Alphabet` and `SymbolList` are defined in terms of in terms of this interface. `Symbol` has two methods. A textual representation is provided by `getName()`, which returns a human-readable string like 'Adenine' or 'gly'. The `getMatches()` returns an `Alphabet` that contains all of the `Symbols` that are valid matches to this one. The matches `Alphabet` will by definition contain the symbol itself, as it must match itself. For a `Symbol` that is ambiguous, such as 'n', this alphabet will contain multiple items. For a `Symbol` that has no ambiguity, such as 'a', this will return an `Alphabet` containing just that single `Symbols`. Gaps are represented by symbols that return an empty alphabet for `getMatches()`, representing the idea that there is literally nothing there, even though space must be reserved for it. Two `Symbol` instances are considered equivalent if their `getMatches()` alphabets contain exactly the same set of `Symbols`. An `Alphabet` does not contain a `Symbol` if there are any members of `getMatches()` that are not also members of the `Alphabet`.

`BasisSymbol` extends `Symbol` and adds the method `getSymbols()` that returns a `List` of `Symbol` instances. Any column of any alignment can be represented as `BasisSymbol` instance, as it is a list of individual symbols, one from each sequence in the alignment. If a `Symbol` comes from some `Alphabet` 'a' that can be represented as the cross product of a list of alphabets, 'A', then it is a `BasisSymbol` if it can itself be represented as a list of `BasisSymbols`. All one-dimensional `Symbol`s are `BasisSymbols`, as clearly an alignment of a single sequence contains columns with single symbols in it.

The codon [a, a, t] is a `BasisSymbol` because it is represented by the list 'a', 'a' and 't' from the DNAxDNAxDNA alphabet. The codons {[a, a, t], [a, c, t]} can be represented as [a, {a, c}, t], which again is a list of three symbols. The codon {[a, a, t], [a, c, t], [a, a, g]} can not be represented as any single list of symbols, so it is not a `BasisSymbol`. However, {[a, a, t], [a, c, t], [a, a, g], [a, c, g]} can be represented as [a, {a, c}, {t, g}], so is a `BasisSymbol`. An `Alphabet` does not contain a `BasisSymbol` if there is any member of `getMatches()` that it does not contain. Additionally, an `Alphabet` does not contain a `BasisSymbol` if it is of a different order to the `Alphabet` (for example, the basis symbol is of length 3, but the alphabet is the product of two other alphabets).

`AtomicSymbol` extends `BasisSymbol` but adds no methods. These symbols actually make up an alphabet, and are never ambiguous. There is an `AtomicSymbol` for 'a' or the codon [a, t, c]. Since `AtomicSymbol` instances can't be ambiguous, `AtomicSymbol` adds the constraints that `getSymbols()` must return a `List` that only contains `AtomicSymbol`s. For the same reason they also add the constraint that `getSymbols()` must return an `Alphabet` that contains exactly one `Symbol`, and that should be the instance itself. Two `AtomicSymbol` instances are considered to be equal if they are referred to by the same Java reference, that is, they are comparable by the == operator. This constraint is not mathematically required, but is necessary to implement efficient algorithms. A Java reference is of the same size as a pointer, and the == operator will have the same overhead as pointer comparison. On many architectures, this will be as efficient as comparing integers or characters.

In practice, most user code never need know that `Symbol` instances can be cast to `BasisSymbol` or `AtomicSymbol` because the API is complete enough that any APIs

can be defined to work with `Symbol` directly and hide any casting inside library code methods. For example, the `TranslationTable` interface defines a method to translate a `Symbol` from one `Alphabet` into a `Symbol` in another `Alphabet` (representing the concept of a function with the domain being the first `Alphabet`, and the codomain being a subset of the second `Alphabet`). This can be implemented by maintaining a table for each `AtomicSymbol` in the source alphabet and the associated `AtomicSymbol` in the target alphabet. Given any `Symbol`, it would first check if it was castable to `AtomicSymbol`. If it is, then the return value can be found directly by looking it up in the table. If it is not, then each `AtomicSymbol` instance in the `getMatches()` alphabet can be translated in turn and a new ambiguous symbol can be made representing this set of translated symbols. In either case, the code calling the translate method need not know anything about the actual type or implementation of the symbol instance.

The gap symbol needs special treatment to avoid various logical problems. The purest version of gap would represent a perfectly empty set, which is dimensionless. Indeed, the EMPTY_ALPHABET constant contains just this entity. The pure gap is a `Symbol` that returns EMPTY_ALPHABET in response to `getMatches()`. All `Alphabet` instances contain this gap. This is because there is no `AtomicSymbol` instance that matches the gap, so there can never be one that matches the gap that is rejected by any `Alphabet` instance.

In addition to the pure gap, there are gap symbols that represent columns in alignments that contain gaps themselves. We can refer to a `BasisSymbol` that is a list containing as [gap]. The gap and [gap] symbols are distinct entities. The pure gap takes up no space in any direction. The [gap] symbol takes up space in one direction. Alphabets like DNAxDNA would contain both gap and [gap, gap], but it would not

contain [gap], as [gap] is a 1-dimensional `BasisSymbol`, and DNAxDNA is two-dimensional. Using this notation, we could represent a column in an alignment between two sequences with a gap in the first sequence as [gap, sym]. Symbols like this have no AtomicSymbol instances that could possibly match them, so their `getMatches()` alphabet is empty. In geometrical terms, this is similar to finding the volume of a solid that has one dimension of size zero. If there is some alphabet that is the cross product of other alphabets, some of which are themselves cross products of other alphabets, the gap symbol respects this. For example, the alphabet DNAx(DNAxDNA) would have the gap symbol [gap,[gap,gap]]. Although this looks complicated, it is in fact necessary to correctly maintain all available information about a `Symbol`. It allows algorithms such as the dynamic programming recursions to distinguish between insertions in each sequence, cells that are at the start or end of one sequence, and cells that lie outside the range of the sequences.

The `Alphabet` interface represents a set of `Symbols`, and can therefore be uniquely represented as a set of `AtomicSymbol` instances. It follows that any ambiguity symbol is a member of an alphabet if its `getMatches()` `Alphabet` is a subset of the alphabet. The `contains(Symbol)` method returns true if the argument is a member of the `Alphabet`, and false otherwise. As a convenience to code that uses `Alphabet`, it also has the method `validate(Symbol)` that throws an `IllegalSymbolException` if the symbol is not contained in the alphabet, and silently returns otherwise. This is in concept similar check to a run-time class cast check.

`Alphabets` have a name retrieved by `getName()`. This is intended to be human-readable. It is also used as a unique identifier for the alphabet, allowing alphabets to

be serialized between different virtual machines and resolve to a single unique instance.

To convert from text to `Symbol` instances, an `Alphabet` can provide access to multiple `SymbolTokenization` instances via the `getTokenization(String)` method. The tokenization registered under the string 'token' will allow `String`s to be parsed into `Symbol`s using some well-known single character codes. Alphabets may provide other tokenizations.

When the product is taken of a `List` of `Alphabet` instances, `getAlphabets()` for the resulting `Alphabet` will returns an equivalent `List`. It follows that `getAlphabets()` returns a `List` of `Alphabet`s that when multiplied together in that order would generate that `Alphabet`. The EMPTY_ALPHABET constant is equivalent to the product of a zero length list. One-dimensional `Alphabet`s such as DNA, return a single element `List` containing themselves. The `Alphabet` DNAxDNAxDNAxPROTEIN would return the list [DNA,DNA,DNA,PROTEIN], and so on.

Two factory methods allow `Symbol` instances to be retrieved from an `Alphabet` while allowing it to maintain internal state and manage memory efficiently. The `getAmbiguity()` method takes a `Set` of `Symbol` instances. It returns a `Symbol` instance that has a `getMatches()` value that contains all of the `AtomicSymbol` instances matching any one of the `Symbol` instances in the `Set`. The `getSymbol()` method takes a `List` of symbols and returns a single `Symbol` that represents the product of these. Both methods validate the input collections to ensure that the resulting `Symbol` is a legal member of the `Alphabet`.

These methods could potentially need to do some fairly involved processing. They must make sure they return the most specific type of symbol possible. If an equivalent symbol exists in the virtual machine, it should return that instance. The `AlphabetManger` class provides several methods to help in this process, simplifying the implementation of `Alphabet`.

`FiniteAlphabet` extends `Alphabet` and represents the case when there are a finite number of `AtomicSymbol` instances that are contained in the `Alphabet`. Because the set is now finite, we can meaningfully define some more methods. The number of `AtomicSymbol` instances in the `Alphabet` is returned by `size()`, and `iterator()` returns an `Iterator` over these. Additionally, it adds the mutator methods `addSymbol()` and `removeSymbol()`, which allow the set of symbols contained to be altered.

Alphabets such as DNA and PROTEIN are represented by instances of `FiniteAlphabet`. There are non-finite `Alphabet` instances for things like the set of all integers and doubles. The `Alphabet` representing all `Symbol`s for integers between 1 and 100 will be a `FiniteAlphabet`. Otherwise, they are non-finite, and just implement `Alphabet`. There are a range of package-private implementations of `Alphabet` and `FiniteAlphabet` that implement these rules. `AlphabetManager` provides the main API for manipulating these entities.

`SymbolList` represents a list of `Symbol` instances from a single `Alphabet`. The `getAlphabet()` method returns the `Alphabet` it is over. The `symbolAt(int)` method returns the `Symbol` at that index. The length of the `SymbolList` is returned by `length()`. Arguments to `symbolAt()` must be between 1 and `length()` inclusive. A portion of the `SymbolList` can be retrieved by invoking `subList(int start, int`

`end)`, where start and end must also be legal indexes. The sub-list returned is inclusive of both start and end.

The default implementations of `SymbolList` in BioJava over finite alphabets all use the 'flyweight' design pattern (Gamma, Helm et al. 1994)[195] to keep memory consumption to a minimum. Internally, the `SymbolList` maintains references to the small number of `Symbol` instances in their `Alphabet`. This means that in a `SymbolList` that is a million DNA symbols in length will be represented as a list of one million references to the four DNA `AtomicSymbol` instances in the DNA `Alphabet`. Because the `SymbolList` interface places no requirements on the actual storage of the data, it is possible to implement many different storage mechanisms. For example there are implementations that fetch portions of the underlying data from files or databases on demand.

Alphabets for DNA, RNA and protein are in use daily with the BioJava toolkit. Additionally, subsets of the alphabet of all double values are used to represent DNA physical properties (such as curvature or flexibility), and higher order alphabets are routinely used to encapsulate everything from multiple-sequence alignments to the results of alignment algorithms to 3-D coordinates. The apparent complexity of the underlying symbol model has more than paid for itself by the vast increase of potential applications now available to objects and algorithms which purely rely on these interfaces, not the underlying data.

2.5 Locations, Sequences and Features

The `Symbol`, `Alphabet` and `SymbolList` APIs were primarily designed to be a good basis for developing algorithms. In contrast, `Sequence` and `Feature` are designed for representing bioinformatics concepts such as database IDs, repeat regions and exons.

`Sequence` represents an entire biological sequence, be it a chromosome, a clone or a primer. A `Feature` represents a region of a `Sequence` that is annotated as being interested for some reason. It may, for example, be a repeat, an exon or a protein active site. The position of a `Feature` is specified by a `Location` object.

The `Location` interface represents an immutable set of indices. A `Location` may be a single index (such as 73), or the range of indices (like [1000..1100]), or all indices that are odd, or any other arbitrary set (for example, {73, [1000..1100]}). `Location` defines the methods `getMin()` and `getMax()` to return the lowest and highest index contained within that `Location`. The method `contains(int)` indicates whether an index is contained. For all locations other than EMTPY_LOCATION, both `getMin()` and `getMax()` are contained by the `Location`.

There are specific implementations for special cases, such as `PointLocation` for a single index and `RangeLocation` for a contiguous range, and `CompoundLocation` for dis-continuous regions. The `equals()` method will return true if two instances contain exactly the same set of indices, regardless of the concrete class of the instances.

The methods `isContiguous()` and `blockIterator()` work together to expose the state of the `Location` without exposing the storage. The `isContiguous()` method returns true if there are no indices above `getMin()` and below `getMax()` that are not contained. The `blockIterator()` method returns an `Iterator` over a minimal set of `Locations` that are guaranteed between them to contain each index exactly once, and are themselves contiguous. For a contiguous `Location`, this will return an `Iterator` that just returns that instance.

There are several methods that compute new `Locations` from old ones; `translate(int dist)`, `intersections(Location l)` and `union(Location)` perform the obvious functions. The methods `overlaps(Location)` and `contains(Location)` return true or false depending on whether the argument overlaps or is entirely contained within the `Location` respectively. Using these operations, it is possible to build arbitrary `Location` instances without ever needing to know how a `Location` is implemented. This makes code using the `Location` APIs very easy to maintain, while enforces ridged encapsulation.

`Locations` are used within the context of the APIs described here. They have, however, been found useful for a range of other applications including bookkeeping to store which pixel indices have been used in GUIs, and also for prime-number searching algorithms.

The `FeatureHolder` interface represents a collection of `Feature` instances. It has methods to count how many features it contains, return an iterator over them, and to return a `FeatureHolder` containing all the Features that match a filter criterion.

There are implementations of `FeatureHolder` that directly store features. Other implementations encapsulate views of over `FeatureHolders` (for example, performing a translation and strand-flip operation) or just store the rule necessary to fetch the underlying data when needed (quite common when implementing adapters to high-latency storage, such as databases).

The `Annotatable` interface specifies one method, `getAnnotation()`, which returns an `Annotation` object. The `Annotation` object is just an associative array (key to value mapping) where arbitrary information can be stored.

`Sequence` extends `FeatureHolder`, making it a container of features. It also extends the `SymbolList` interface so that it can represent the primary sequence. In addition, it adds a name and URI property for naming the sequence uniquely and also extends `Annotatable` so that arbitrary information that pertains to the entire sequence can be stored.

There are implementations of `Sequence` that store the sequence and features in memory. Other implementations include those that manipulate whole-chromosome assemblies or data from Ensembl (Hubbard, Barker et al. 2002) and DAS. The interface-centric design means that an implementation of `Sequence` that suits a particular situation can usually be trivially composed from a suitable implementation of `SymbolList`, `Annotation` and `FeatureHolder`. As Java does not support multiple inheritance of implementations, this is achieved by storing references to objects implementing each of these interfaces and explicitly forwarding method invocations as needed.

Other than the effort required to initially enter the code that forwards method calls, this actually has some benefits over inheritance. Firstly, the implementing objects may in some cases be expensive to initialize or use a lot of memory. As they are private state of, and not directly part of (by inheritance) the implementation, they can be lazily instantiated. Secondly, it is possible to choose a specific implementation class at run-time, for example, by choosing an implementation of `Annotation` optimized for efficiently storing very small numbers of values, or for retrieving values associated with a very large number of properties.

`Feature` extends `FeatureHolder` and `Annotatable`. In addition, `Feature` has source, type, location, parent sequence and symbol properties. Source and Type are

equivalent to the GFF source and type fields. Source should represent the program or process that produced the evidence for the feature (such as a particular gene finder), and Type should indicate what the feature is meant to represent, such as CDS, or transcription start site. The `Location` represents which region of the `Sequence` the `Feature` is attached to. The parent is the `FeatureHolder` that directly contains this `Feature`. The Sequence property always refers to the `Sequence` that ultimately contains the `Feature`. As `Feature` extends `FeatureHolder`, it is possible to build up arbitrary hierarchies of features (but always as a tree). For example, a gene feature may contain zero or more child features that represent transcription factor binding sites or perhaps exons. The parent of the exon would be the gene feature, and the parent of the gene would be the sequence. However, both exon and gene objects will return the same `Sequence` for `getSequence()`. Lastly, the symbols property returns a `SymbolList` that represents the `Symbols` contained within the `Feature`. The exact semantics of this method is left up to the `Feature` implementation. Sub-interfaces of `Feature` are provided which add more specific properties such as strand information, frame and protein translations.

So that the feature hierarchy on sequences can be modified, there must be an API for adding features to other features and to sequences. So that a given implementation of `Sequence` and `Feature` can maintain implementation integrity, there must be some sort of factory method (Gamma, Helm et al. 1994)[107] defined in the interfaces. The original implementations had methods like `createFeature()`, `createStrandedFeature()` or `createExon()`, but as the number of interfaces grew, it became obvious that this would not scale well as the interface would have to be modified every time another type of feature was added.

This was solved by using a single `createFeature()` method that takes a polymorphic argument of type `Feataure.Template`. The template has public fields that hold the properties of the feature to make, such as location, type, source and the like. This mirrors the memento design pattern (Gamma, Helm et al. 1994)[283]. In each interface that extends `Feature`, a public static inner class extends `Feature.Template` called by convention Template. For example, to instantiate a `StrandedFeature`, you invoke the `createFeature()` method with an instance of `StrandedFeature.Template` as the only argument. It is then the responsibility of the `Sequence` and `Feature` implementation to create a `StrandedFeature` implementation with the same information as the template. If the particular `Sequence` implementation can't provide an appropriate implementation of `Feature`, it should either instantiate the closest one it can and put the missing information into the annotation bundle, or throw an exception. This approach allows sequence implementations to support an arbitrary sub-set of the available feature types without requiring the feature creation interface to grow in complexity with the number of feature types defined.

Sequences and features represent the bulk of information manipulated by most applications. By designing the APIs from the foundation to support polymorphism and encapsulation, we have produced a design that allows the underlying data to be represented in any one of a number of different ways. There are implementations that are backed by relational databases (Ensembl and BioSQL adaptors), CORBA (BioCorba adaptors) and by web services (DAS and XEMBL clients) in addition to those using Java objects directly. The feature creation API provides a uniform and easy to implement way to create features conforming to a range of interfaces with a range of different concrete implementations. The result is that developers can interact

with a single API and have access to a wide variety of different information without needing to know anything about the implementation details of how this is achieved.

2.6 Probability Distributions and Hidden Markov Models

Hidden Markov Models (HMMs) (see Section 1.3.2) are a popular method of analysing biological sequences. BioJava contains APIs for representing and working with probabilistic HMMs. This includes code for representing models, as well as implementations of the common dynamic-programming (DP) algorithms for evaluating alternative state-paths and training model parameters. All of these APIs build upon the `Symbol`, `Alphabet` and `SymbolList` APIs (Section 2.4), allowing them to be applied without change to the wide range of signal types these data-structures can be used to represent.

To model HMMs effectively, it is useful to provide a mechanism for representing probability distributions. There are a wide range of other contexts within which probability distributions can be used, such as in modelling weight matrix columns, so to aid in their reuse there is a separate Java package dedicated to their representation and implementation.

The `Distribution` interface encapsulates a probability distribution over an `Alphabet`. The method `getWeight(Symbol)` returns the current probability of observing the symbol from the probability distribution. This is notionally equivalent to integrating or summing a probability distribution out over the range of all `AtomicSymbol` instances in the symbol's `getMatches()` `Alphabet`. The manufacture of distributions is usually performed by a `DistributionFactory` object. This allows particular implementations of `Distribution` to be tailored to a particular `Alphabet` without client code needing to know the details. For example, the default

implementation of the factory returns `Distribution` implementations that use either linear lookup or binary search lookups based upon which is most time-efficient for the alphabet size. In addition, `OrderNDistribution` extends the Distribution interface, and defines that it will be a probability distribution over one alphabet conditional upon another. For example, given the `Alphabet` DNAxDNA (containing all ordered pairs of nucleotides), an `OrderNDistribution` could be built that was four independent `Distribution`s over the second `Alphabet` conditioned upon the first (i.e. the probability of the second nucleotide appearing given that we knew what the first one was).

Background database probabilities of the amino acids in Swiss-Prot (Boeckmann, Bairoch et al. 2003) could be represented as a probability distribution over the PROTEIN alphabet. The probabilities could be estimated by counting the frequencies of each amino acid in the database and then normalizing these counts to give a probability distribution.

The `MarkovModel` interface encapsulates state-emitting HMMs. `MarkovModel`s contain one or more `State` objects. The `State` interface extends `AtomicSymbol`. `EmissionState` specializes `State` by having an associated emission `Distribution`. In generative model terms, `EmissionState`s emit the symbols within sequences. `DotState` extends `State`, and represent non-emitting, silent states. These are useful for rationalising the architecture of models.

The `MarkovModel` interface has a `stateAlphabet()` property that returns a `FiniteAlphabet` containing every `State` in the model. It also has an `emissionAlphabet()` property that is the `Alphabet` that matches the `Alphabet` of the `Distribution` objects associated with the `EmissionState` instances. In addition,

there is a method `getWeights(State)` that returns the transition probabilities from a `State` as a `Distribution`. The `Alphabet` of the `Distribution` will be a sub-set of the states `Alphabet`, representing every `State` that is reachable from that `State`.

`MarkovModel` also has a property `getHeads()` that represents how many `SymbolList` instances are aligned to each other and the model. A single-head model emits a single `SymbolList` of sequence and a single `SymbolList` of `State`s. These co-linear lists of symbols and states can be used to label a sequence, which may, for example, mark up features like repeat regions, protein domains or exon boundaries. Models with two heads perform pair-wise alignment. These can be used to align pairs of sequences based upon evolutionary relationships, or to find portions of two sequences that are more similar than would be expected by chance. Models with three or more heads align that number of `SymbolList` instances to one another and label the alignment with the states used.

The `EmissionState` interface defines one other property named advance, which is an array of integers (usually 0 or 1) that indicate how much each head of the model is advanced by the emission. For example, in pairwise alignments, the states that emit aligned regions will advance in both directions, having an advance property of [1, 1], where as the insert states will have an advance of [1, 0] and the delete states will have an advance of [0, 1]. The `Distributions` associated with emission states should emit gap `BasisSymbols` that have a `BasisSymbol` in each dimension that is 1, and a gap in each dimension that is 0. Gaps are used to represent the concept that there is a gap in the list of symbols for that dimension, or equivalently that although the global index has advanced, the index of the underlying data being viewed has not.

The `MarkovModel` interface is the data-structure that defines how some sequences could be emitted. These HMMs are purely data, and have no algorithms associated with them. The recursions that align sequences given a `MarkovModel` are defined by the `DP` interface. `DP` defines methods to calculate the forwards, backwards and Viterbi recursions (see Equation 1-13). In addition, it defines a method to generate sequences from the model. The efficient implementation of these recursions depends upon the structure of the model and the number of heads the model has.

To hide implementation detail from the user, the `DPFactory` interface defines how to get a `DP` implementation for a given model. For models with one or two heads, there is a `DP` factory implementation that returns `DP` implementations that invoke interpreters. For two head models, there is also a `DP` implementation that generates Java byte code optimized to the architecture of a particular model. The `DP` compiler outperforms the `DP` interpreter significantly, particularly for models that contain many states that have transitions from only one source state. The interpreter is more suited to situations where the model architecture is being altered, as the compiler would have to produce new code each time the model architecture is modified.

For pairwise alignment, using the notation of Equation 1-13, $\bar{i}$ is a 2-tuple of the index for the first and second sequences respectively. We can impose a partial ordering upon the set of 2-tuples that follows the natural ordering of each component. To calculate the cell at $\bar{i}$, we must first have calculated all cells that are before $\bar{i}$. For the 2-dimensional case, this means calculating the cells at $\bar{i}-(1,0)$, $\bar{i}-(0,1)$ and $\bar{i}-(1,1)$. The naive way to ensure this is to construct an in-memory matrix to store results, and to perform a nested loop over two index variables starting at 0 and going to the first and second sequence length respectively. There are many ways to loop

over all possible values of $\bar{i}$ while guaranteeing this ordering. As long as all of the values that are needed for uncalculated cells to be calculated are present, it is not necessary to store any of the other values.

Assuming that the entire dynamic programming matrix is not required, we can write a space-optimised implementation of pairwise DP that uses space proportional to the length of the shortest sequence. Given an index a into the first sequence (the shortest) and b into the second (the longest), we can have an outer loop over the values of b. A single row of the dynamic programming matrix (indexed by a) containing the results of the previous iteration (at $b-1$). Then, a new row can be calculated for the row at b. At the end of the iteration, the column at $b-1$ and be discarded, and that at b becomes the array used as the known results for the next iteration (at $b+1$).

The back pointer structure is a matrix of the same shape as the Viterbi matrix, but storing the state used to reach that cell. This contains all the information necessary to trace back from the final cell to the beginning of the alignment to retrieve the highest scoring alignment. However, this introduces a space cost proportional to the product of the sequence lengths. It is clear that many sub-optimal paths will exist through the matrixes that are not needed for the eventual trace back.

Instead of holding a reference to the previous state in the matrix, BioJava stores a `BackPointer` object. This stores a reference to the previous `BackPointer`, a step-wise score and reference to the `State` associated with that position. Because `BackPointer` instances refer directly to the previous entry in the chain, there is no need to store the entire matrix in memory explicitly. The Java virtual machine will take care of garbage-collecting all `BackPointer` instances that are not reachable from the current states. It is then only necessary to explicitly hold in memory the

BackPointer objects associated with the cells that still have uncalculated dependencies.

In this way, the memory cost for the back pointer data-structures can be reduced to something that is at a maximum proportional to the memory used to store the values of the dynamic programming matrix, and which converges to something proportional to the length of the final alignment (which can never be longer than the sum of the lengths of the sequences). The BackPointers will form a directed a-cyclic graph. The trace back path must be from one of the leaves of this graph to the root. While calculating this directed acyclic graph (DAG), the garbage collector will drop entire branches from memory when they are no longer reachable. The more fully connected the model is, the quicker the BackPointer graph will converge towards being linear on alignment length.

Space-saving versions of the Forwards and Backwards recursions can be similarly constructed. Since these algorithms consider all possible paths, there is no need to consider the BackPointer data structures, so these algorithms require memory proportional to the length of the shortest sequence and the number of states only.

This allows very large pair-wise alignment problems to be considered without memory resources becoming the limiting factor. Clearly, this does not remove the need to evaluate every part of the recursions, so the algorithms still scale computationally on the product of the lengths of the sequences being aligned.

The parameters of a Distribution (and by extension the emission and transition probabilities in an HMM) can be estimated using the DistributionTrainer interface. This provides a transactional framework for associating observed counts

with a `Distribution`, for aggregating these, normalizing them and resetting them to zero. Given labelled data, parameters are estimated by adding whole counts to the `DistributionTrainer` proportional to the observations and then invoking the train method to update the `Distribution` parameters.

In many cases, a collection of distributions will need to be trained simultaneously. The `DistributionTrainerContext` interface encapsulates such a set. This allows all of the distributions within an HMM to be trained simultaneously. Since `Distribution` is an interface, there will often be cases when implementations do not actually store the values directly, but rather perform some calculation on the parameters of another `Distribution`. For example, there is an implementation of `Distribution` in BioJava that takes an underlying `Distribution` and a table that maps input `Symbol` instances to a `Symbol` for an underlying distribution. This allows, for instance, a `Distribution` over DNA symbols to have emission probabilities equal to those of the complementary symbols in another `Distribution`. When the distributions are registered with a `DistributionTrainerContext`, the implementations will ensure that counts for the complementary view are routed on to the underlying `Distribution` instance, and once the context is asked to train all the parameters from the aggregated counts, the complementary view will reflect the new parameters of the newly trained distribution.

Training distribution parameters via `DistributionTrainerContext` allows very complex parameterisation of models to be explored, both in terms of the emission probabilities and for the transitions as well, without altering the dynamic programming recursions or the routines used to collect observation counts.

HMMs have been constructed with this API to model 3-D DNA structures (`BasisSymbols` with one dimension per physical property and `Distributions` that model multinomial Gaussians over these properties), align pairs of protein secondary structure, find transcription factor binding sites, perform GIBBS sampling of expression data, find eukaryotic and prokaryotic promoters, as well as a host of other tasks.

2.7 Query

2.7.1 Motivations

Fairly early on in the use of the `Feature` interfaces, there was the need to find features of a particular type, or with particular properties, or some combination thereof. Initially we started adding many `getFooByBar()` methods, but it quickly became apparent that this would not scale.

2.7.2 Initial Implementation

After reading the Dragon compiler book (Aho, Sethi et al. 1985), we developed a small language for describing constraints for accepting or rejecting feature types, and added the method `filter(FeatureFilter aFilter, boolean recurse)` to the `FeatureHolder` interface. The `FeatureFilter` interface has the single method `accept(Feature)` which returns true if the feature is to be included in a return-set and false otherwise. There are implementations of `FeatureFilter` for accepting features based upon their properties (type, location, annotations and the like). There are also several logical filters. For example, `FeatureFilter.And` is an implementation of filter that will accept a feature if two other filters both accept it. There are implementations for the logical Operations 'and', 'or', 'not' and 'nand'. Using these logical Operations and the basic filters, it is possible to build up quite

sophisticated constraints. The 'recurse' flag in the filter method indicates whether to apply the filter to the current collection of features only or whether to apply it to those features and all features that they contain recursively. This provides coarse-grain control over how to navigate the feature hierarchy.

The filtering language allows collections of features to optimise the processing of requests as they can interrogate the query to find portions that they can easily process. For example, a given `FeatureHolder` implementation may know that it only contains features of a particular type. It can then optimally handle any filter expression that contains a `ByType` expression by just comparing the two types and either accepting or rejecting the entire set of features. This is much cheaper than comparing every feature in turn with the filter expression. Filters are used in nearly all library and script code that manipulates features. For data-specific implementations such as the DAS or Ensembl bindings, the ability to compare filters can be used to implement reasonable lazy-fetching strategies to avoid loading unnecessary information into memory from high-latency storage (the web, or an SQL database for example). For GUIs, the rules for deciding which features to display can be stored in these flexible filter objects and modified at will.

2.7.3 Limitations of This System

This scheme has served us well over the last two years. However, there are several shortcomings with this approach. The first one is that it can only be applied to features. To extend this to all BioJava objects would require many sets of filters to be written, each with rules about how to interpret them. In addition, in practice it would be nice to be able to retrieve sequences from a `SequenceDB` instance based upon whether they do or don't contain a given type of `Feature`. This would require the

ability to specify filters that span multiple object types. The other main shortcoming is the way that the recursion through the feature hierarchy is performed. For example, when retrieving features of a particular type within a region of a human chromosome we must recurse down through each level of the assembly pruning it as we go according to region and then search for features of that type at each level. This process is not easy to represent as a single filter. In practice, we end up constructing recursive function calls that each do non-recursive filters to prune the selection by location, find the features of the appropriate type to return, and recurs down to each feature with children.

Because the filter statement does not represent the entire process of finding the features, it is impossible to perform optimal searching and data-caching strategies for these complex cases. This causes potential inefficiencies to creep in to an otherwise elegant system.

To address these issues, we are evaluating a range of approaches for modelling complex queries, including finite state machines, ontology languages and graph grammars (refs).

2.8 Recent Developments

Since late 2001, BioJava has continued to be developed, expanding far beyond the original group of two individuals. There are now over thirty developers, five of which form the core development team. In this time, existing APIs have been consolidated, and new ones have been added. In this section we will discuss some of the areas where I have personally been the primary developer, as well as some of the functionality which the community has contributed.

The three primary areas of personal contribution are in the design and implementation of the tag-value parser framework, flat-file indexing, and a constraints-based type system for `Annotation` objects.

Major community contributions include improvements to the `FeatureFilter` language, the 'change hub' mechanism for managing large numbers of change listeners, bit-packed sequences, parsers for blast (and other sequence similarity search formats) and an emerging API for representing and manipulating ontologies. To a greater or lesser extent I have had personal involvement in each of these, but the bulk of the design or implementation has been undertaken by others.

2.8.1 The Tag-Value Parser

A large proportion of the data analysed by bioinformaticians is stored in text files. Commonly, these are structured as lists of records. Each record is composed from one or more lines that contain a tag specifying its type and an associated value. For example, EMBL entry files have entries separated by lines consisting of '//', and each entry has one or more lines with a two letter line-type identifier code (such as 'AC', 'OC' or 'FT') with a value present in columns 6 to 80. Genbank files have a similar structure, but in this case the record separator is '///' and the different line types are identified by full names (such as 'ACCESSION', 'ORGANISM' or 'FEATURE'). There are a large number of file formats that closely resemble either EMBL or Genbank files, but contain different tags and represent different types of information, such as classes of enzymes (Bairoch 2000), taxonomies (Benson, Karsch-Mizrachi et al. 2003) and protein families (Falquet, Pagni et al. 2002).

In our experience, developing custom parsers for these file formats is an error-prone task. One system that has implemented a general approach to parsing biological flat

files is the SRS system with its language Icarus[22]. Here we describe a similarly generic framework for parsing these files within BioJava.

The tag-value framework is an attempt to provide a unified way to abstract out the common parts of the parsing task (such as recognizing record boundaries and dividing lines of text into tags and values) while allowing the exact details to be customised as needed. The approach we took was to use a mixture of the strategy (Gamma, Helm et al. 1994)[315] design pattern and liberal use of listeners. Strategies are used to encapsulate the variable portion of a process into an interface on its own, so that the unchanging portion can handle the unchanging functionality and delegate to the strategy where needed. All data is treated as Java `Object` instances rather than `String` instances, allowing the same framework to work un-changed on on-textual information.

Over all, the flow of parsing events is very similar to that in the Simple API for XML (SAX[23]). In the XML analogy, the text files are like XML files, the tag-value listeners are like SAX events, and the `Annotation` API is the equivalent of the Document Object Model (DOM[24]). There are also similarities with the Boulder IO package[25], as well as the way that the BioPerl SearchIO has been designed.

---

[22] We have been unable to find documentation about icarus on the LION bioscience web site. However, the EBI is currently providing documentation, which can be found at http://srs.ebi.ac.uk/doc/icarus.pdf

[23] The SAX standard is coordinated through the http://www.saxproject.org/ web site

[24] The DOM specification can be found at http://www.w3.org/DOM/

[25] BoulderIO is described at http://stein.cshl.org/software/boulder/

The class `Parser` has a single method `read(BufferedReader, TagValueParser, TagValueListener)` that reads all of the text from the buffered reader, uses the tag-value parser to process this into tags and values, and informs the tag-value listener of these pairs. For users of the API, this is the main method that they would invoke.

The `TagValueParser` interface encapsulates the process of splitting each line of input into a tag and a value, and also of deciding if the tag is new or not. If the tag is different from that on the previous line, then the parser assumes that it is new. In the case where it is the same, the tag-value parser can indicate that it should be treated as a new instance of that tag, rather than as an additional value for the current tag. For example, rather than SWISS-PROT comments being treated as just one series of values for a single comment tag, the tag-value parser could force a new comment tag event to be fired for each logical block of comments. There are implementations of `TagValueParser` that split lines into fixed-width areas (with two pre-built instances, for files formatted similarly to EMBL and Genbank), and one that splits according to a regular expression.

The `TagValueListener` interface has five methods. These are all invoked by a `Parser` instance, and it is the `Parser` that is responsible for ensuring correct nesting of these method invocations. The two methods `startRecord()` and `endRecord()` signal that records have started and ended respectively. All other events are emitted within the scope of this pair of events. The `startTag(Object)` and `endTag()` methods indicate that a tag has been started and ended respectively. These are called within the scope of the record. Tags are never directly nested. That is, for every `startTag()`, there is never a directly nested `startTag()` invocation, and for every `startTag()` there is exactly one matching `endTag()`. The `value(TagValueContext,`

`Object)` method is used to inform the listener of values associated with the current tag. The `value()` method is only ever invoked within the context of a tag, and never directly within the context of the record. For each value associated with a tag, there will be a separate invocation of `value()`, and it is up to the listener how this should be interpreted.

Values can be replaced with `Objects` that are not `String` instances. For example, while parsing an EMBL entry, the lines relating to organism information could be transformed by a listener into a single taxonomy value. It is common to transform textual representations of things like URLs and Enzyme Classification (EC) numbers into light-weight objects. This greatly enhances the richness of the data consumed by the ultimate listener.

Some of these tag-value file formats have embedded sub-documents. For example, EMBL and Genbank files have an embedded feature table document. The tag-value framework supports these by using the context passed in as the first argument to `value()`. The listener can uses the `pushParser(TagValueParser, TagValueListener)` method to indicate to the `Parser` that all values of the current tag should themselves be split into tag and value pairs. The pushed tag-value parser will be used to split the values of these lines, and the results will be passed onto the pushed listener. Once the outer tag ends, the pushed parser and listener pair are popped back off the processing stack, and the original listener will be informed of an `endTag()` event as normal. The listener pushed will receive the full set of start/end record and tag events associated with the sub-document, and may itself choose to push new listeners for embedded documents.

This framework allows flexible processing of files into event streams. However, it is useful to further process these events. To support this there are a range of listener implementations that wrap other listeners, and pass on altered evens. For example, the accession lines of EMBL and Genbank files can contain a list of accession numbers, separated by semi-colons. A listener would receive one value event for each accession line. The data becomes easier to interpret if one value event can be produced for each accession number. A `RegexSplitter` instance could be used to recognize each portion of the accession line that is an accession, and then fire one value event to the wrapped listener for each accession.

The `ValueChanger` listener implementation is the class responsible for changing values associated with particular tags. It is responsible for either replacing a value with some other value, or for firing off multiple values. Again, the strategy pattern is used, in this case to factor out the mapping between tags and actions into a separate class named `ChangeTable`. A `ChangeTable` instance maintains a table of which actions are associated with which tags. This greatly promotes code reuse and modularisation. The `ValueChanger` code just manages the flow of events. The actions themselves are trivial to implement as little Java classes (often in practice as anonymous classes). We have found that this kind of composition and parameterisation is far superior to inheritance-based methods of customizing behaviour.

The `TagMapper` listener is used to systematically replace tags with other tags. For example, it would be possible to configure a `TagMapper` instance to map all EMBL tag names to Genbank tag names. This allows event streams to be transmuted into those accepted by standard listeners and factory objects. For example, by

transforming reference information in to tags and values that resemble those emitted from the EMBL parser, the same reference handlers used for EMBL processing can be reused.

Using the built in tag-value classes, and by supplementing these where needed with some application-specific code, it is possible to rapidly develop parsers for tag-value formats of nearly any kind, and transform the information in these files into that required for a particular application, while achieving a very high degree of code reuse.

2.8.2 Flat File Indexing

A related problem to that of parsing these files is that of retrieving one entry among potentially the many hundreds of thousands of entries in a single or multiple files. It was decided by members of the OBF that it would be useful for all of the projects to share a mechanism for indexing these files. There exist a number of indexing strategies (for example, emblcd[26]). However, these tend to pose problems when accessed from multiple different languages and on multiple platforms as they are binary file formats. The OBDA flat file indexing specification[27] defines an indexing strategy that just uses plain text files to store the indices.

BioJava contains a full implementation of this specification, allowing a wide range of file types to be indexed, and for individual records to be fetched in time

---

[26] Applications for manipulating embl CD index files can be found at http://www.hgmp.mrc.ac.uk/Software/EMBOSS/Apps/dbiflat.html

[27] See http://cvs.biojava.org/cgi-bin/viewcvs/viewcvs.cgi/obda-specs/flatfile/indexing.txt?rev=HEAD&cvsroot=obf-common&content-type=text/vnd.viewcvs-markup for the most recent version of the specification.7

proportional to the cost of a binary search through the index. Records can be retrieved either by primary ID or by secondary IDs. The BioJava implementation is wholly compatible with the other OBF implementations (in Perl, Python and `C`).

Sequence files can be indexed using the standard BioJava classes for reading sequences from a stream. Additionally, any format that can be parsed with the tag-value framework can be indexed. In the future, we will be continuing to enhance support for secondary IDs, and provide simple APIs to allow different flat-file formats to be linked to one another by IDs in a manner similar to SRS[28].

2.8.3 Annotation Types

Since the beginning, BioJava has supported free-form associations of keys and values through the `Annotation` interface. Ironically, many applications need to be able to guarantee that certain property keys will be present in an `Annotation`, or that certain values will be present. The project started to gain a lot of repetitive and error-prone code that first checked an annotation to see what properties and values it had, and then acted accordingly.

In order to reduce the need to write this error-prone and repetitive code, we chose to develop a dynamic type system for `Annotation` instances, based around the new `AnnotationType` interface. The main two methods are `instanceOf(Annotation)` and `subTypeOf(AnnotationType)`. Both compare the argument to the current type. The `instanceOf()` method returns true if the argument is an instance of the type, and `subTypeOf()` returns true if the argument is a sub-type of the type. An `Annotation` is

[28] Mode information about SRS can be found at the http://www.lionbioscience.com/solutions/products/srs web site

an instance of an `AnnotationType` on the basis of what properties and values it has. One type is a sub-type of a super-type if every annotation that is an instance of the sub-type is also an instance of the super-type. Two `AnnotationType` instances are equivalent if exactly the same set of `Annotation` instances are accepted by the `instanceOf()` methods of both types.

Restrictions are placed upon the range of values that can be associated with properties by using instances of the `CollectionConstraint` interface. This has two main methods. The `accept(Object)` method returns true if the argument is acceptable to the constraint. The `subConstraintOf(CollectionConstraint)` method returns true if all items acceptable by the sub-constraint are also acceptable to the super-constraint. The `AnnotationType` `instanceOf()` and `subTypeOf()` methods are implemented purely by using these methods. There are implementations of CollectionConstraint that perform the normal logical operations, as well as checking properties of the item under consideration.

There are many utility methods in `AnnotationTools` that deal with the common operations that may be performed upon `AnnotationType` and `Annotation` instances. This use of the façade design pattern (Gamma, Helm et al. 1994)[185] insulates users of the API from its necessary complexities. `AnnotationTools` implements a wide variety of logical operations upon `AnnotationType` directly, such as computing types that are the logical union, intersection and difference of two types. It is very common to compare the results of these to the `AnnotationType` constants that accept or reject every `Annotation`. Additionally, it can be used to generate new `Annotation` instances from an old one and a type, for example, by retaining or removing all keys defined by the type.

The AnnotationType APIs differ from the way the Java type system works. In Java, every `Object` maintains a reference to its `Class`. This `Class` maintains references to all `Class`es that it inherits from, both implemented interfaces and extended classes (the Java introspection APIs use the same type to represent both classes and interfaces). With `AnnotationType`, `Annotation` instances maintain no such reference. As properties are added and removed, or the associated values are altered, the `Annotation` may change which annotation types it is an instance of. Code that wishes to check types should always use the `AnnotationType.instanceOf()` method. We tend to think of annotations as being 'castable to' an annotation type, rather than inheriting from or implementing them.

The `AnnotationType` interface works synergistically with annotations and the tag-value parsers. If the tag-value framework is like SAX and the `Annotation` API is like DOM, then `AnnotationType` is like XML-SCHEMA[29]. Annotation types are also used extensively in the recently enhanced implementation of feature filters.

2.8.4 Enhanced Feature Filters

The `FeatureFilter` APIs have since been developed by the community into a fully functional constraint language for `Feature` hierarchies. In addition to adding implementations for nearly all conceivable feature properties, there are now implementations that accept or reject a feature based upon the type of the annotation associated with the feature. Additionally, there is now much finer grained control of searches through the hierarchy, using filters like `ByAncestor` and `HasChild`.

---

[29] The XML Schema and related standards can be found at http://www.w3.org/XML/Schema

There is a façade class named `FilterTools` that implements many common operations upon feature filters. This includes composing new filters that are the union, intersection or difference between two filters and a range of factory methods. The results of these are often compared to the `FeatureFilter` constants that accept or reject all features. Another very common operation is to compare two `FeatureFilter` instances to see if they accept a disjoint set of features.

A range of `FeatureHolder` implementations now publish FeatureFilter instances as schemas describing what features they contain. Given a query, it is possible to efficiently see if the query is disjoint from the schema and potentially avoid comparing the contained features to the filter.

Some Feature and Sequence implementations now look at the FeatureFilter instances being passed into the filter() method, and check the filter for known types of annotation. For example, if a FeatureFilter is used to filter Ensembl, and it is constraining upon the "ensemble_id" property in the feature's annotation bundle, the Ensembl code will be able to recognize this and do optimized database lookups rather than looping over all possible feature instances.

For database and high-latency applications, such as Ensembl and DAS, this constraints-based language has allowed the existing Sequence and Feature APIs to scale gracefully to queries that potentially scan many hundreds of thousands of entities. By careful comparison of the filters with known schemas, and by introspecting the filters for constraints that can be optimised, we experience performance comparable to that for special case code that manually plans search strategies.

2.8.5 Change Hubs

Another bottle-neck upon scalability was the implementation of how change listeners were registered with resources such as database objects. Sequence databases that contain mutable implementations of `Sequence` need to behave as if the complete network is in place to forward events from every one of the sequences it contains. In the case of database implementations like BioSQL and Ensembl, there may be anywhere from a few hundred to several hundreds of thousands of sequence and feature instances that theoretically need to be listened to. However, in a typical application, only a few of these are ever directly accessible in memory.

In these special cases, there are implementations of the Changeability support classes which we call "change hubs" that maintain data-structures to keep track of which listeners logically exist. As each sequence is instantiated in memory, the change hub registers the required listeners to it. As the sequence goes out of scope, it takes care of removing the listeners. In this way, users of these databases can appear to be listening indirectly to a vast number of objects, while the implementation cost in terms of memory, and the time needed for event notification, can be kept to a minimum.

This is one of the many examples of where designing BioJava from the start in terms of interfaces has allowed us to drop in a complex replacement for some standard functionality without altering the APIs exposed to users.

2.8.6 Bit Packed Sequences

As it became more common to work with complete genomic information, the original implementation of `SymbolList` became impractical. It stores references to `Symbol` instances in an array. References in Java are the same size as a pointer. On

32-bit platforms, this is a clear overhead compared to storing bytes (8 bits), and this is even more pronounced on 64-bit platforms.

To allow very large sequences to be loaded into memory, an API was developed for mapping between `Symbol` instances and bit patterns. The `Packing` interface encapsulates the mapping between all `Symbols` in an `Alphabet` and unique bit patterns. To encode all atomic symbols, the bit patterns must be wide enough to encode the size of the alphabet. For example, the DNA alphabet has four members. This can be packed into two bits. The PROTEIN alphabet has 20 members. This will require 6 bits as 5 bits can only represent 16 combinations, but 6 bits can represent 32. To encode all symbols, including the ambiguities, the bit pattern will have one element per atomic symbol in the alphabet. This allows the presence or absence of that atomic symbol to be indicated. DNA therefore requires 4 bits, and PROTEIN requires 20 bits.

For small alphabets, like DNA and RNA, the memory saving associated with packing the sequence is considerable (2 bits vs. 32 or 64 bits). The performance penalty is approximately a factor of two compared to the implementation that accesses references directly for linear scans. However, the code that is shared between the packed and unpacked implementations has been tuned now to be over three times faster than it originally was, so that the current packed implementation is in fact faster than the original unpacked implementation.

Long symbol lists are implemented by storing a `List` of fixed-length symbol list instances. When a `symbolAt()` request comes in to the symbol list, it calculates which child list it is in, extracts that symbol list and passes the request on (after adjusting the index accordingly). One benefit of this is that if a sub-list has been taken of a region,

and that region falls totally within the range of one of the child symbol lists, the sub-list will only maintain a reference to this child symbol list, allowing the large one to be garbage collected once it goes out of scope.

Another benefit is that different child lists can be implemented using different `SymbolList` implementations. Most sequences present today have a very small number of ambiguity symbols, and when they are present, they are usually runs of 'n' characters. Child lists that have no ambiguity at all can be packed using the most efficient packing possible. Child lists that do have ambiguity can be packed using an ambiguity-capable packing.

The bit-packing APIs have facilitated the implementation of a pure-java implementation of SSAHA (Ning, Cox et al. 2001), as well as making it feasible to store complete human chromosomes in memory. It demonstrates again the flexibility afforded by interface-based design. No methods in the `Alphabet` or `SymbolList` interfaces needed to be modified, meaning that all existing applications benefit from these improvements without alteration.

2.9 Conclusions

The BioJava APIs outlined here are designed to be extremely flexible, while imposing minimal restrictions on how the interfaces are implemented. This is achieved by pervasively using Java interfaces rather than abstract classes to define APIs, and leveraging nested exceptions to handle errors. Potentially variable behaviour is systematically encapsulated by strategy objects. The Changeability API allows programmers to maintain tight control over which object may be modified and under what circumstances this will be allowed, while also facilitating the synchronization of objects' state and simultaneously enforcing the principal of strong

object encapsulation. The Symbol and Location APIs provide examples of how careful object modelling can make software disproportionately powerful by ensuring that the interfaces have a complete but minimal set of operations that allow for all conceivable uses of the objects. The `DistributionFactory` and `DPFactory` interfaces demonstrate how tailor-made implementations of interfaces can be instantiated without the users of APIs needing to know about these implementations. The creation of features by using `Feature.Template` instances demonstrates how two-dimensional polymorphism (interface and implementation) can be implemented without pushing responsibility for data-integrity to the users of the API. The `FeatureFilter` and `AnnotationType` APIs demonstrate how data structures can be queried efficiently without violating encapsulation.

Because of the strongly interface-centric design, it is fairly easy to view underlying data in several forms by defining only the transformation to be applied. For example, `OrderNSymbolList` views an underlying symbol list as the nth order view. A 1st order view of a DNA sequence would produce a `SymbolList` with the alphabet DNAxDNA, containing symbols that represent each overlapping pair of symbols from the original sequence. Similarly, distributions can be constructed that represent a translated view of another distribution, such as the complementary distribution. This paradigm allows very elegant data-structures to be built up without duplicating either the underlying data or the code that performs view transformations. Indeed, BioJava strongly discourages data duplication. As an extreme example, to reverse-complement an entire chromosome, BioJava would construct a `ComplementarySymbolList` that views a `ReverseSymbolList` that views the underlying `SymbolList` for the chromosome. The total additional cost in memory for complementing the chromosome is two Java object instances, rather than the memory for the entire

Chapter 3 HMMs for whole *Plasmodium Falciparum* Chromosomes

3.1 Introduction

Observation of chromosomes in a variety of organisms appears to show that they are composed of a number of distinct blocks. For example, there are the banding patterns observed in condensed eukaryotic chromosomes (Rooney 2001). With the primary sequence of these chromosomes becoming available it is now possible to investigate what relationship if any there is between these patterns and the sequence.

By using unsupervised learning techniques it is possible to look for natural patterns in the sequence without being biased by prior expectations. We can then compare these natural patterns with the annotated biological function to look for correlations. If the chromosomes are constructed from blocks that have one of a small number of sequence composition biases, it should be possible to estimate both the number of and the compositional bias for each distinct bias and use these to partition the chromosome into regions. In the general case, the chromosome could be modelled as being made up of regions of DNA which each have a reasonably constant sequence composition but which noticeably vary in composition from their neighbours.

The compositional bias parameters and the likely order in which blocks follow one another can be estimated using Hidden Markov Models (HMMs) (see Section 1.3.2). In contrast to the complex HMM methods commonly employed for modelling biological sequences, such as gene finders, our models do not need to be concerned with the fine structure of the DNA, concentrating instead on large-scale chromosomal structures.

It has been observed that gross sequence content correlates particularly strongly with function in the Malarial parasite *Plasmodium Falciparum*. The chromosomes as a whole have a very high ratio of AT to GC. However, since coding for amino acids require all nucleotides to be used, exons tend to have a slightly lower ratio (Escalante, Lal et al. 1998). Annotators use sequence composition plots as a tool to aid annotation. Since this is particularly useful in the *P. Falciparum* annotation process (K. Rutherford, personal communication) this genome was selected as a target for investigation using these approaches.

P. Falciparum has a genome estimated to be about 30[30] megabases (mb) in length, divided into 14 chromosomes (Pollack, Katzen et al. 1982). The genome exhibits a strongly biased AT/GC ratio with an overall (A + T) content estimated at 82 %. Recently, the complete sequence of chromosomes two and three have been sequenced and published (Bowman, Lawson et al. 1999; Gardner, Tettelin et al. 1999)[31]. The telomeric regions of chromosomes two and three are similar in structure, containing a shared pattern of terminal telomeric repeats followed by the repeats R-CG7 and rep20, a member of the *var* gene family, the R-FA3 repeat and finally a *riffin* gene. This arrangement of repeats and genes appears to be functional, promoting shuffling of the sub-telomeric regions between multiple chromosomes (Figueiredo, Freitas-Junior et al. 2002).

---

[30] The total size of the genome has since been found to be closer to 23 mb in length Gardner, M. J., N. Hall, et al. (2002). "Genome sequence of the human malaria parasite Plasmodium falciparum." Nature **419**(6906): 498-511..

[31] Since this time, the entire genome of *P. Falciparum* has been sequenced Ibid..

By eye, it is possible to identify regions of chromosome 3 with extreme base composition. Some of these are clearly correlated with biologically important features. In addition to the GC enrichment associated with exons, there is a region with extreme AT content (95-100%) that is believed to be the centromere (Hall, Pain et al. 2002). It is interesting to speculate as to how many types of sequence composition exist within Malaria chromosomes, or even whether the chromosome can meaningfully be grouped into regions that have one of a small number of compositional biases, or are in fact part of a continuum.

The BioJava HMM APIs are very flexible and allow many different architectures and parameter sets to be evaluated rapidly. The representation of the underlying alphabets in BioJava enables us to reuse architectures for different representations without recoding the core recursions. This makes them an ideal tool for investigating this type of open-ended question.

3.2 Simple HMM Architectures

3.2.1 Methods

A simple HMM was constructed using the BioJava HMM APIs (Section 2.6) with two states each with independent emission distributions. This was expected to segregate the chromosome into regions of high and low AT/GC ratio. A second model was generated with four independent states expected to segregate the chromosome into regions of relatively very low, low, high and very high AT/GC ratio. In both cases, these models were fully connected (transitions existed between all states). Transitions and emissions parameters were initiated to random values, but with the constraint that the transition from any state to itself was initialised to a value approximately 1000 times more likely than the transition to any other state. All model

scores are presented in units of log probability due to the extreme dynamic range of these probabilities. These models were trained using Baum-Welch with sampling (as described in Section 1.3.2).

3.2.2 Results

The two-state model reached a stable set of parameters within a very few cycles. The log likelihood remained almost constant from cycle 40 to completion at cycle 1214 (–1246786 at cycle 40, with mean –1246786 between cycles 40 and 1214). The Viterbi state paths from the model at cycles 40 and 1000 are 98.6 % identical. The model with four states showed similar convergence behaviour (data not shown).

The emission probabilities of the model with two states were complementary rather than being segregated into high and low GC. Over multiple training sessions with different initial parameters, the model with four states learned two distinct sets of model parameters.

Both four state models contained a pair of states that were similar to the states in the two state model. This pair of states aligned to the major part of the chromosome. The other two states of the four state model trained differently.

In the first set of model parameters, the two additional states aligned to the chromosome ends (telomeric regions) and were complementary to each other, i.e. for each telomere one state aligned to one strand and the other to the reverse complement of it.

In the second set of model parameters, the two additional states instead modelled a strong first order relationship in the telomeres. Specifically, one state modelled 'A' rich regions and the other modelled 'T' rich regions. Frequently these 'regions' were

only single nucleotides in length. Transition probabilities favoured them moving from one to another. The other pair of states modelled the internal regions as before.

3.3 HMM Architectures with Complementary Emission Distributions

The above results demonstrate that the chromosome must be considered in terms of being a double-stranded DNA molecule rather than as a single-stranded sequence. In particular, if there is a block with a characteristic sequence composition on one strand, this, by definition, implies a block with the complementary distribution on the other strand. This pair of states should be modelling a single set of parameters. To achieve this we developed a `Distribution` that implements a complementary view onto another `Distribution`. A pair of states can then be added to the model, one with the forward strand distribution and one with its complement. We call these complementary states pair-states. During training, all counts associated with the complementary distribution are first un-complemented and then forwarded as counts to the forward-strand distribution. This guarantees that the total number of parameters is minimized and that all available evidence for emissions is used during training.

3.3.1 Methods

Models were constructed with two, three, four or five pair-states (4, 6, 8 and 10 total states respectively). During training, all emission probabilities and all transition probabilities were initially set to random values, with transitions from each state to itself initially being approximately 1000 times more likely than any other transitions. Each model was then trained using Baum-Welch with sampling, as described above, as well as by Baum-Welch, using the sequence of Malaria chromosome 3. Training was stopped after 100 cycles due to a combination of computational constraints and the observation that models appear to converge before cycle 100. The different

models were then aligned to both chromosome 3 and chromosome 2 without additional training.

Models with more than 5 pair-states were not trained as both the memory and computational requirements for the estimation of training parameters becomes prohibitive. The space required is approximately equal to length_of_sequence x number_of_states x size_of(double), which for large sequences with many states quickly reaches the limits of a machine with hundreds of megabytes.

3.3.2 Results

Training using Baulm-Welch with sampling exhibited quicker convergence properties than Baulm-Welch, and was also computationally less expensive due to the decreased number of counts which needed to be summed. Multiple training runs with sampling produced models with more similar parameters and alignment scores than with Baulm-Welch training (data not shown).

We then considered a representative from the replicates of the two, three and four pair-state models. The Viterbi paths at 20 and 100 cycles for the two, three and four state-pair models differed by 0.23 %, 0.45 % and 1.41 % respectively. This indicates that by cycle 100, the models were not changing significantly in their predictions. The model with five pair-states did not use one pair of states at all, indicating that this family of models could only distinguish four types of gross genomic content, and is therefore not discussed further.

In all cases, some transition probabilities in the trained models have moved greatly from their original values, and the most used states have emission probabilities that lie close to the ratios found in the chromosome (Figure 3-1).

The Viterbi paths for all three pair-state models at cycle 100 against chromosome 3 (Figure 3-2) show use of paired states at the beginning and end of the chromosome, and a similar banding pattern of states within the chromosome. In all three alignments, a single state emits the first 276 bases of the chromosome. The corresponding complementary state then emits the final 186 (±2) bases of the chromosome. This corresponds to the regions of sequenced telomere. In addition, in all three models, a single pair of complementary states emits the majority of the body of the chromosome. The models with more states show additional features, such as the appearance of a band near the ends of the chromosome that resembles telomeric sequence, and blocks of sequence corresponding to repeat elements. Not all states were used by the more complex models. For example, the three pair-state model learned two telomeric states, two internal states and a final state that matches a region within the genes PFC005w and PCFC1120c. This state did not use the complement of this final state anywhere.

The four pair-state model has corresponding states for each of these regions and two complementary states that match a region between the telomeres and the *var* genes. These overlap significantly with the repeat elements rep20, rep11 and R-CG7, and show striking similarities with the state-paths for chromosome 3. Again, the telomeres have been correctly identified, and the exons on each strand seem to segregate with the two main states. In addition, the regions near the telomeres are predicted to have a very similar structure, including a telomeric-like section within the *var* genes, and the use of states that overlap the repeat elements.

In the four state-pair model, the coding region state pair has one state associated with each strand. 86% of all bases in exons on the positive strands are matched by the

first state, and only 14% by second. 94% of all bases in exons on the negative strand are matched by the first state and 5% by the second. Overall, these states predict the strand correctly 90% of the time on a per-nucleotide basis. Most errors are made on the boundaries between genes on opposite strands.

The Viterbi state path for chromosome 2 (Figure 3-3) shows striking similarities with those obtained for chromosome 3 (Figure 3-2). This is evidence that the two chromosomes share a common architecture. Labellings of randomised sequences do not show these similarities in patterns (data not shown). Therefore, we believe that the models have indeed learned some general properties of malarial chromosomes.

It is possible that the consistent structures predicted at the beginning and end of each chromosome is an artefact of transition probabilities associated with entering and exiting the model. To test this, artificial chromosome sequences were constructed. The first half of the chromosome was appended to the second half so that the central regions of the sequence were now at the ends, and the ends of the sequence were now in the centre. State-paths were predicted using the same models as before. The regions at the ends of the artificial sequences were labelled with the states associated with the body of the chromosome, and the regions corresponding to the telomeres now located in the centre of the sequence were labelled with the telomere-associated state-pair. This indicates that the models are making predictions on the basis of the sequences, and not any edge-effect artefacts.

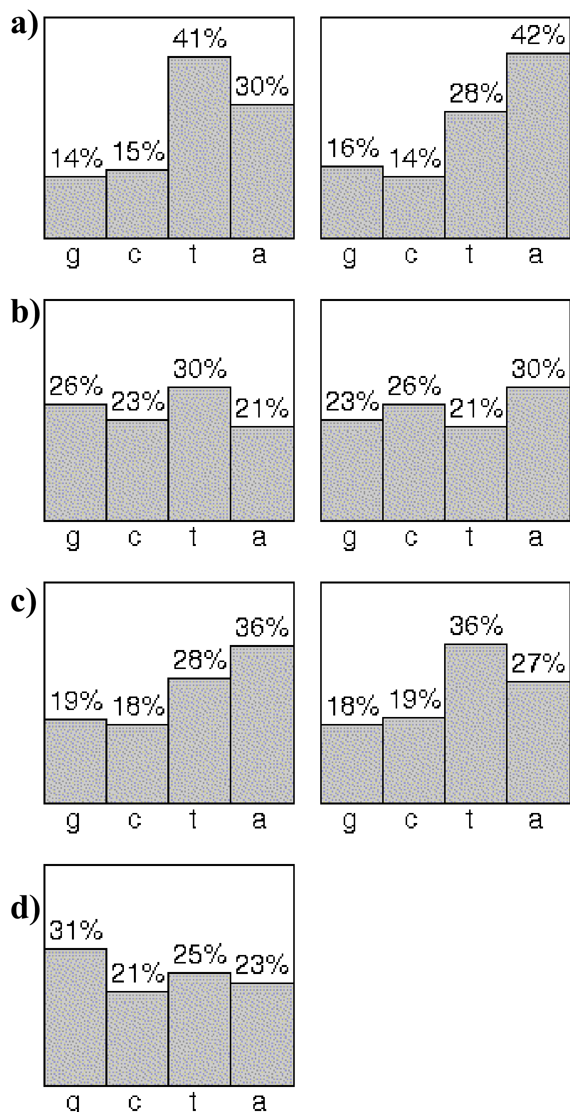

Figure 3-1 Emission probabilities for the four pair-state model

Each row shows a state-pair with complementary emission probabilities. They match **a)** chromosome body; left and right associated with (-) and (+) strand exons respectively **b)** telomere-like sequence; left and right associated with telomeres at the right and left of the chromosome respectively **c)** near-telomere repeat associated regions **d)** (G + C) rich region in the *var* genes (only one of this state-pair is used).

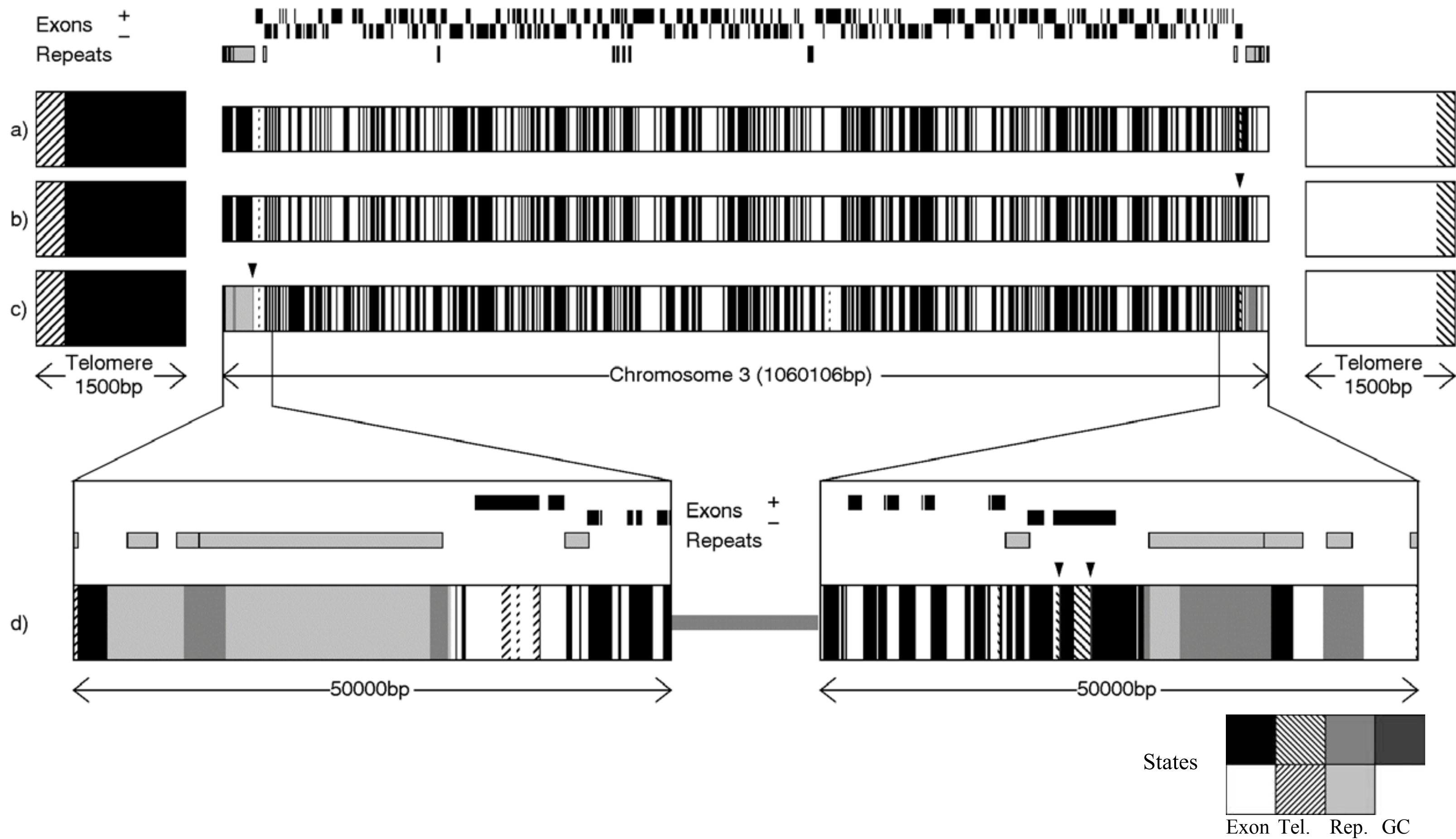

Figure 3-2 Diagram of the *P. Falciparum* chromosome 3 and the state paths through three models

(legend continued on next page)

Sections **a**, **b** and **c** represent the state paths of the two, three and four state-pair models respectively across the entire chromosome with insets to the left and right showing the extreme telomeric region. The relative positions of exons and repeat elements are indicated above these diagrams. Section **d** shows an enlarged view of the state paths for the first and last 50,000 bp of the chromosome, with the corresponding exons and repeat elements above. Within each diagram, a different shading pattern is used for each state, as indicated by the key (Exon - exon-related, Tel. - telomeric-like, Rep. - repeat-associated, GC - high (G + C) content). Arrows above the diagrams indicate the positions of narrow regions that may not be easily visible.

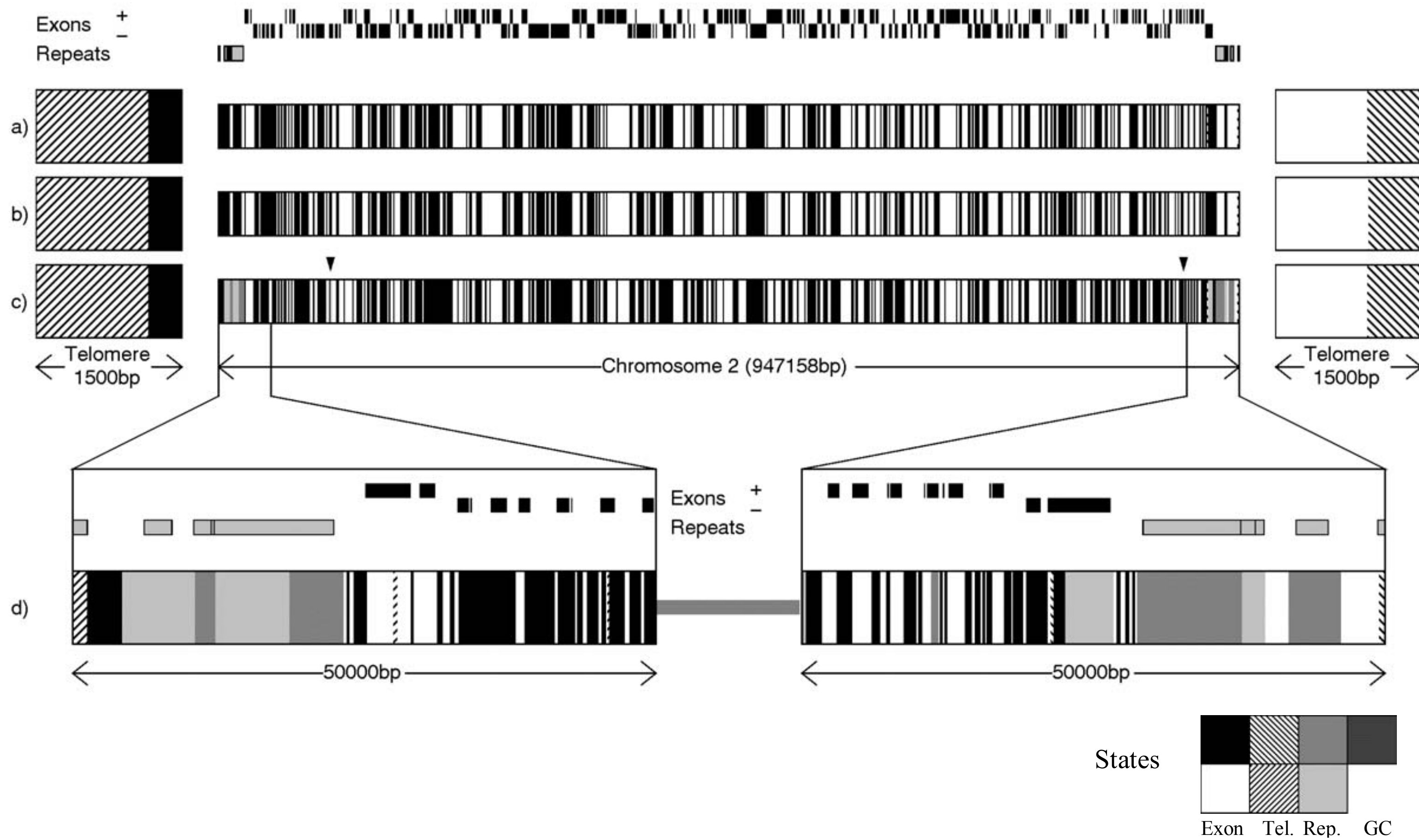

Figure 3-3 Diagram of the *P. Falciparum* chromosome 2 and the state paths through three models (legend continued on next page)

Sections **a**, **b** and **c** represent the state paths of the two, three and four state-pair models respectively across the entire chromosome with insets to the left and right showing the extreme telomeric region. The relative positions of exons and repeat elements are indicated above these diagrams. Section **d** shows an enlarged view of the state paths for the first and last 50,000 bp of the chromosome, with the corresponding exons and repeat elements above. Within each diagram, a different shading pattern is used for each state, as indicated by the key (Exon - exon-related, Tel. - telomeric-like, Rep. - repeat-associated, GC - high (G + C) content). Arrows above the diagrams indicate the positions of narrow features that may not be visible at this scale.

3.4 First Order HMMs with Time-Reversible Transition Probabilities

The model with four independent distributions in some cases learned a pair of states that crudely represented a 1st order distribution (encapsulating dinucleotides). To explore this further, models were constructed that contained emission `Distribution` objects encapsulating a 1st order Markov process.

The 0th order model used the third pair-state to identify a high-G region. However, the transition probabilities learned for this model only allowed one of the pair of states to be used. This was due to the lack of association between the transition probabilities. The re-architecting process required to introduce higher order emission probabilities also provided an opportunity to constrain the transition probabilities such that the resulting HMM is truly time-reversible (if the sequence being analyzed is played back in reverse with the appropriate complementation, it would induce a state labelling that is the reverse-complement of the forward state labelling). The time-reversed transition probabilities should in theory remove the kind of artefacts observed in the 0th order model's third pair-state.

3.4.1 Methods

The chromosome was viewed through an `NthOrderSymbolList` instance to translate it into all overlapping pairs of symbols, and the HMM emission alphabet was set to DNAxDNA.

The 1st order emission distributions presented additional challenges to ensure that they were correct estimates in both the forward and reverse directions. The probability of observing a given dinucleotide is defined as being the probability of observing the second nucleotide conditioned upon the first. That is, there is a 0th order probability distribution over each second nucleotide that is chosen according to the identity of the

first nucleotide. If the distribution associated with the complementary state is calculated by simply reverse-complementing the dinucleotide and finding its probability in the original distribution, this will not be a true probability distribution (the sum over all probabilities given all dinucleotides starting with a given nucleotide will not be guaranteed to be 1). This is because in this case we are effectively conditioning upon the second nucleotide rather than the first. This causes the models to be non-probabilistic and the training algorithms to fail.

We address this by reverse-complement the table of observations and then re-normalize to give the complementary 1st order probabilities. During training, a standard `DistributionTrainer` is registered with the forward-strand probability distribution. The reverse-complement distribution registers a `DistributionTrainer` that forwards all counts on to this after reverse-complementing the dinucleotide. Probabilities for the forward distribution are estimated as normal and those for the reverse distribution are estimated by normalizing the reverse-complemented counts.

This scheme ensures that all available information is pooled (both evidence for forward and reverse strand are aggregated) and that the result is a strand-reversible probabilistic Markov process. As a concrete example, given the short sequence 'AATGCGT' we can estimate both a forward 1st order distribution, that would produce this and a reverse-strand 1st order distribution, that would produce 'ACGCATT' with an equal probability using the counts in Table 3-1 and a suitable normalization (such as pseudocounts). It is clear from this example that the probability of observing a dinucleotide is not equivalent to observing its reverse-complement (for example, AG is half in the forward strand, but CT is not observed at all in the reverse strand).

Table 3-1 Forward-strand and reverse-strand counts

	A	G	C	T	Sum
A	1			1	2
G			1	1	2
C		1			1
T		1			1

	A	G	C	T	Sum
A			1		1
G			1		1
C	1	1			2
T	1			1	2

The transition distributions present a more complex problem. The first naïve approach was to constrain the transition probabilities of a reverse-strand state to be the transition probability from the forward-strand state to the complement of the destination. This does yield a probabilistic model. However, it is not fully time-reversible. This is because if we consider both strands, the model effectively treats entry to a forward-strand state as being equivalent to exiting a reverse-strand state.

This was again addressed by estimating the transition probabilities from tables of counts. However, we run into a problem that prevents us using the same `DistributionTrainer` solution as for the 1st order probabilities. Table 3-2 enumerates every possible transition from state 'a' to state 'b' given that neither, one, or both may be complemented (indicated as a' or b' respectively).

Table 3-2 State-transitions and their reverse-complements

Forward	Reverse Complement
a-b	b'-a'
a-b'	b-a'
a'-b	b'-a
a'-b'	b-a

For three of the four cases, either the forward or reverse-complement forms start with a forward-strand state. These can all use the reverse-complement forward-state counts to calculate the backward-state probabilities. However, in one case (a'-b:b'-a), there is no count associated purely with the forward-strand process. In the naive probability model described above, this case is considered interchangeable with (a-b':b-a'). However, the two are clearly distinct transitions. This issue did not arise for the emission probabilities as we only considered the cases of a-b, or b'-a', which are a well-behaved subset of all the interactions in Table 3-2 (in particular a-b:b'-a').

The problematic transitions do not arise for more restrictive model architectures for which forward and reverse model regions are separated by an a-directional region. During training, a table of counts for all pair-wise combinations of states was kept, and while collecting observations, the count was split into two parts, which were then forwarded to each count cell representing the two possible time-reversed transitions. Then, during training, the distribution was estimated by normalizing the aggregates of each of the two time-reversed transitions.

3.4.2 Results

Models with 2, 3, 4 or 5 pairs of states were trained on chromosome 3 of *P. Falciparum* using Baum-Welch training until the forwards probability did not vary by more than $e^{0.01}$ between two cycles (changes of > 0.01 relative to scores in the range of tens of thousands). The models took 116, 103, 77 and 90 cycles respectively to converge. Models with more states were again not trained due to the memory and computational constraints.

The transition probabilities for all models are dominated by state-to-self transitions. Emission probabilities for all models (Figure 3-4) show a progression in complexity

with the number of state-pairs available. The additional distributions seem to model additional sub-types of sequence, and in every case, each of the distributions in the simpler models are represented in the more complex models. This indicates that the additional available complexity is being used to model distinct populations of sequences. This is in contrast to the 0th order model, which could model no more than four compositional biases. Presumably, the 1st order probability distributions are capturing some more biologically relevant information. It is also interesting in that each model was trained entirely independently with different starting parameters but learned very similar final parameters. This is good evidence that the models are learning some legitimate signals embedded within the chromosomal sequence rather than using the extra parameters in an arbitrary manner to memorise the training sequence.

The entire chromosome was classified into the following biological feature types; exon, intron, repeat and other, using the annotation associated with the malarial chromosome. This classification was then projected onto the state labelling from the 5 pair-state model (shown graphically in Figure 3-5 and Figure 3-6 for chromosomes 3 and 2 respectively). From this, a count of the number of times a particular state and feature are co-located was calculated (Figure 3-7). These counts show dramatic trends for certain features and states to be associated with one another. There are clearly two state-pairs (3± and 5±) associated with exons. States 2± are also associated to some degree with the 'other' category while States 4± accounts for the majority of repeats. Figure 3-8 and Figure 3-9 are normalized views of these counts for the 5 pair-state model representing the conditional probabilities of observing a particular state given that the feature is known, and observing a given feature given that a state is known.

From Figure 3-8 it is clear that if a region is an exon, the states 3± and 5± together account for nearly the entire feature (> 93 % for + strand, > 94 % for –strand). Introns do not have a single state associated, but show a predisposition towards the pair 2± (2- preferred over 2+ for forward strand introns and 2+ preferred over 2- for backward strand introns). Indeed, the predispositions in forward and reverse strand introns appear to show a distinctly strand-dependant pattern despite there not being a single indicator state. This indicates that the introns contain important strand-dependant information. Repeats are associated with the states 4±. The other category most closely resembles the average of the intron distributions. It is interesting that the repeat distribution seems not to include a large proportion of states 1±, despite these being found in introns and 'other'.

The most striking feature of Figure 3-9 is that almost all states are associated primarily with only one feature type. The second observation is that no state is predictive of introns. States 1± and 2± are associated with 'other'. States 3± and 5± are associated with exons. States 4± are associated with repeat regions.

We can see from Figure 3-7 how as state pairs were added, the correlation between states and features altered. In the 2 pair-state model, exons are only labelled by states 2±, but these states are also frequently found labelling 'other', and accounts for almost all repeats. In the 3 pair-state model, the exons and the repeats are modelled by their own states (3±), while 2± remain the major 'other' states. In the 4 pair-state model, states 4± now take on the role of specifically modelling the repeats. Finally, in the 5 pair-state model, states 5± model a sub-set of exons. Clearly, as more states are added to the models, they are making finer distinctions over how to model the chromosome.

Figure 3-5 and Figure 3-6 are graphical representations of the state-paths of the five pair-state model to chromosomes 3 and 2 of Malaria respectively. Again, from these figures, the co-localisation of some feature types with biological features are clear to see, particularly at the extreme ends of the chromosomes. These results also demonstrate how as the complexity of the models increase, finer distinctions in the assignments are identified.

Figure 3-7 displays the frequency with which different states align to each of the different biological feature classes. In Figure 3-8, this data has been normalized to give the observed probability of any given state given a particular type of feature. This gives an indication of how strongly a given state labelling of a region of chromosome indicates a particular biological function for that region. In Figure 3-9, this same data has been normalized to give the observed probability of any feature type given a particular state. This indicates how predictive each state is of the different feature classes.

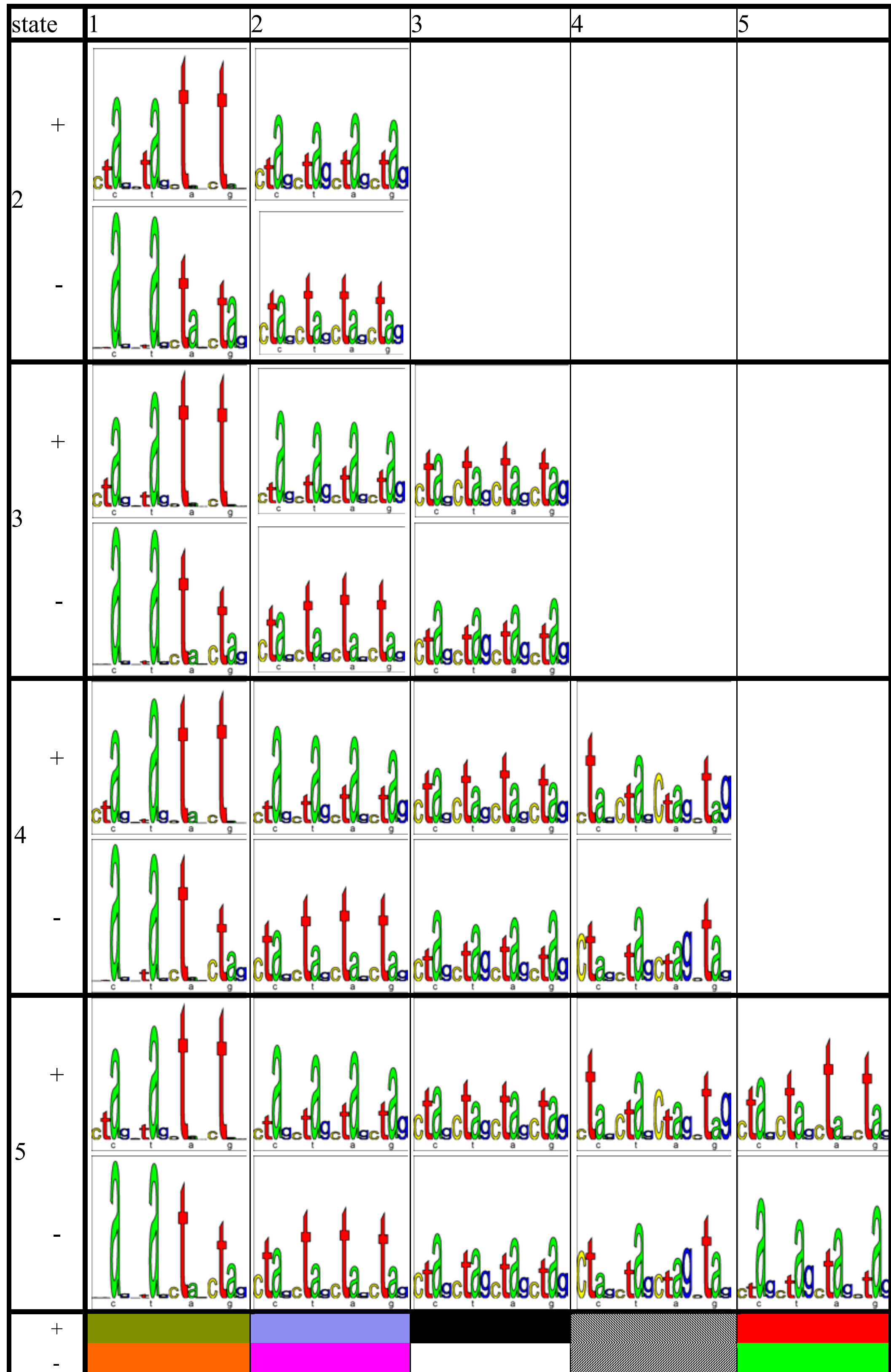

Figure 3-4 Emission Spectrums for all Pair-State Models (legend continued on next page)

Sections for the 2-5 pair-state models contain two rows, the first contains graphs of the 1st order emission probabilities and the second contains graphs of the reverse-strand emission probabilities. The emission probabilities for each model are arranged so that those that appear similar are in the same column (1-5). The final area displays a key that associates the states with colours in the whole-chromosome diagrams Figure 3-5 and Figure 3-6.

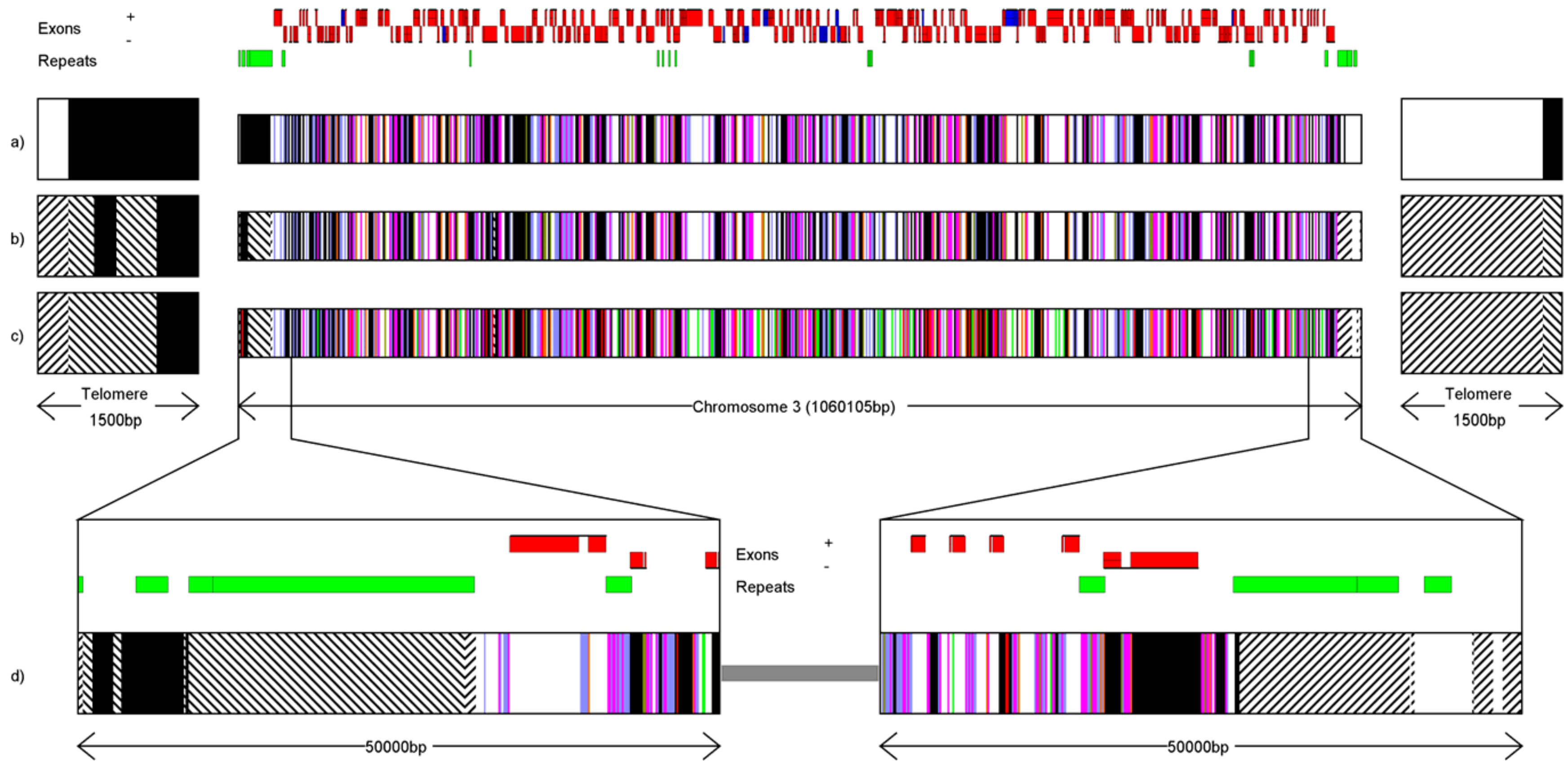

Figure 3-5 Diagram of the alignments of the 3,4 and 5 state-pair models to Malaria chromosome 3

(legend continues on next page)

Lines **a**, **b** and **c** display the alignments of the 3,4 and 5 state-pair models respectively. The colours are as in the key in Figure 3-4. The red exons belong to 'normal' genes. The blue exons belong to 'bob' genes (Bowman, Lawson et al. 1999).

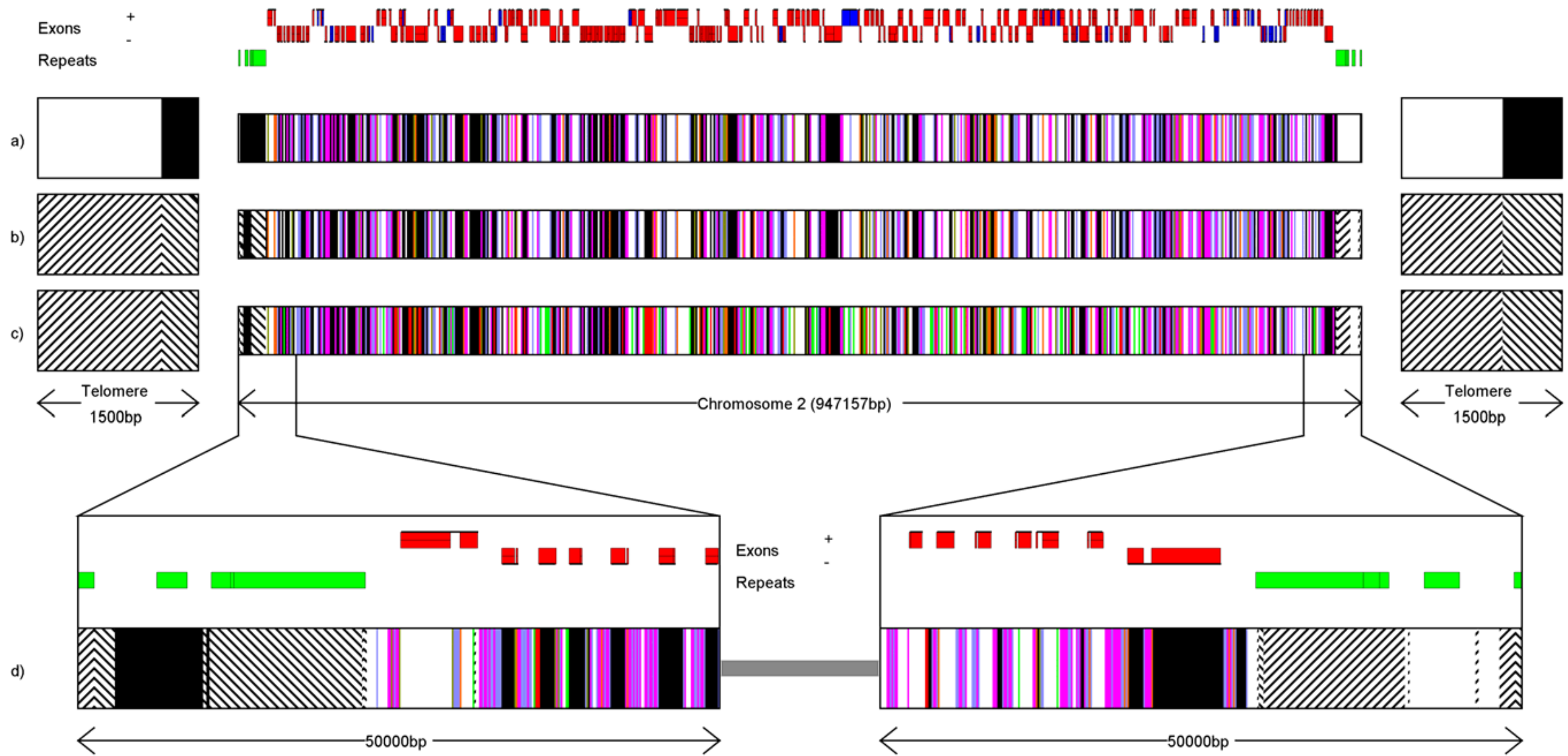

Figure 3-6 Diagram of the alignments of the 3,4 and 5 state-pair models to Malaria chromosome 2

(legend continues on next page)

Lines **a**, **b** and **c** display the alignments of the 3, 4 and 5 state-pair models respectively. The colours are as in the key in Figure 3-4. The red exons belong to 'normal' genes. The blue exons belong to 'bob' genes (Bowman, Lawson et al. 1999).

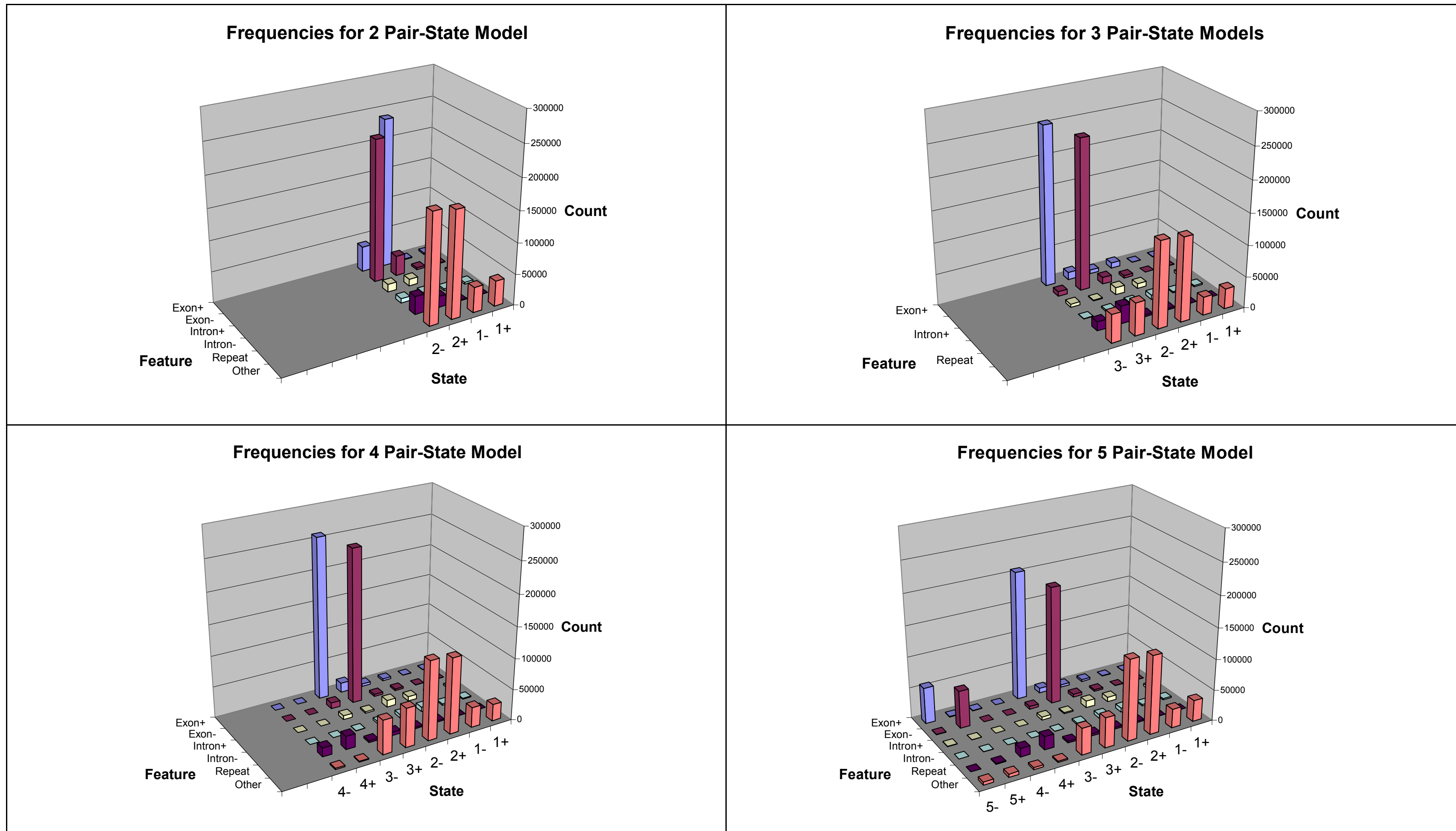

Figure 3-7 Counts for Biological Feature and States for the 2-5 Pair-State Models

(legend continues on next page)

The state labels are consistent with the labels in Figure 3-4. Each model in turn was used to label chromosome 3. The state labelling was then compared to the location of known genes and repeats.

Error! Not a valid link.

Figure 3-8 Normalized Counts of States for Biological Features

This bar chart displays the data in Figure 4-8 grouped by biological feature. For each feature, the probability of observing a given state is displayed as a bar. Exons are primarily accounted for by states 3±, with states 5 aligning to only approximately one fifth of exon sequence. No other biological feature class is predicted so clearly by any state.

Error! Not a valid link.

Figure 3-9 Normalized counts of Biological Features for States

This bar chart displays the data in Figure 4-8 grouped by state. For each state, the probability of observing a given feature is displayed as a bar. States 1+, 1-, 2+ and 2- are specific for the Other category. States 3+, 3-, 5+ and 5- are specific to exons States 4+ and 4- are specific to repeats.

3.5 Discussion

In order to investigate the gross structure of malarial chromosomes, we explored a number of different HMM architectures using the BioJava HMM APIs. The initial model contained just two states. It was trained to classify every base within the Malaria chromosome as being emitted by one or other of these states. The transition probabilities were set to initial values that favoured a single state emitting a long region of the chromosome. The belief was that this model would segregate the chromosome into high and low AT/GC content. After training, the emission spectrums of the two states were very close to being complementary. One interpretation of this is that the underlying biological process learned was strand-dependant, so that in effect the model reflected a single probability distribution, but learned it once for each strand.

This observation lead to the construction pair-state HMMs with pairs of states that emit nucleotides according to complementary distributions. The two pair-state model revealed that the telomeric regions were distinct from the internal chromosomal sequence, and that the body of the chromosome aligned to a single pair of states which flipped between one another. This pair of states appears to correspond to the positions at which Malaria utilizes one strand or the other for coding exons, predicting the strand with an accuracy of 90 %. This is surprisingly accurate, given the extreme simplicity of the model.

The more complicated models additionally predict a feature resembling telomeric sequence followed by a small region of sequence that is distinguished from the rest of the chromosome by its very high (G + C) content (52 %). This interesting pattern is visible only in the genes PFC005w and PCFC1120c, which are putative members of

the *var* family (Bowman, Lawson et al. 1999), involved in evading the host immune system. There is some evidence that *var* genes are subject to epigenetic control and undergo frequent intragenic recombination (Corcoran, Thompson et al. 1988), so the telomeric-like fragments within these genes may be the remnants of chromosomal rearrangement events resulting in the shuffling of these sub-telomeric regions.

The models trained on chromosome 3 were aligned without further training to *P. Falciparum* chromosome 2, to check whether the models had learned features specific chromosome 3, or more general features of Malarial chromosomes. Without further training, the models correctly recognise the telomeres, predict the exon directions and also identified the telomere-associated repeats in chromosome 2. In addition, the *var* genes on chromosome 2 appear to contain a band of telomeric sequence in the corresponding locations to the *var* genes located on chromosome 3. In chromosome 2, the band of high (G + C) content appears not to be present.

The blocks of telomeric base composition within, and beyond the *var* genes, may be a relic of recent recombination events between these genes and other telomeric *var* loci. Other *var* loci also appear to share this feature (data not shown), although these types of model may not be appropriate for analysing short sequences. The high (G + C) region found in the chromosome 3 alignments may be specific to that chromosome as none of the other *var* genes analysed shares this feature. It was not possible to train simple pair-state models that used more than four pairs of states, which is evidence that the models were not over-fitting the training data, and were characterizing real information about the chromosomes.

In addition to the observation that both strands of the chromosome must be considered, the original model indicated that some of the processes observed were not

easily modelled by 0th order probabilities. This inspired the creation of the fully time-reversible 1st order models. These models were able to learn more subtle signals that were associated with or were indicators of exons (+ and – strand), introns (again + and – strand), repeat elements and 'other' (assumed to be intergenic sequence). These models were capable of consistently learning the same signals given different initial training parameters and different numbers of paired states. Additionally, they were able to learn additional and more complex signals as the number of parameters was increased. The most interesting feature of the 5 pair-state model is the sub-division of exons into those with high and low adenine content (states 3± and 5± respectively). This does not coincide with any obvious properties of the genes.

None of these models learned a state associated with the putative centromere, which has been predicted to lie in a region which is almost entirely (A + T) in composition (Bowman, Lawson et al. 1999). However, the centromere is a comparatively small structure that may not be distinctively different from the already extreme A/T bias of the chromosome in general. The 0th order model did model a very small region of high G+C content, but this had sequence-composition characteristics that are radically different to those associated with the other states. It is possible that a 1st order models with more states would have recognized the centromere.

All of these models were trained using unsupervised learning techniques, and had no supplied data to indicate the location or type of biological features. However, all of these models have learned signals that are co-located with biologically significant structures. Given the relative simplicity of the models, this is clearly a potentially powerful method.

3.6 Future Directions

At the time this work was done the model size was limited due to the physical memory required to store training parameters for large sequences. With newer machines with greater physical memory it has now become practical to extend this work to consider more states and larger sequences, such as human chromosomes. It would also be interesting to look at orders greater than one. However, to train models with large numbers of transitions and high-order emission states would most likely require a more sophisticated regularization framework and possibly a more complex representation of the HMM than simple pseudo-counts or these probability parameterized finites state machines can afford.

The memory requirements for the dynamic-programming matrices used during training scales linearly with the length of the training sequences, and also with the number of states in the model. On the computer hard ware used in this study, this becomes prohibitive for sequences that exceed more than a megabase in length, and for models with more than ten states.

One solution would be to calculate one matrix completely, and then calculate each column of the other in turn using the space-saving implementation of the recursion, adding counts associated with each completed row of the matrix as we go. However, one of the matrices must still be held in memory, so this still scales in proportion to the length of the sequence, allowing us only to double the training sequence length, or the number of states.

Another solution would be to calculate the space-saving version of one recursion, and as each matrix row is completed, calculate the other recursion back to that point. Although the memory required for this is trivial, the computation will scale by the

square of the sequence length. This is likely to become prohibitive even quicker than the memory constraints of the above approaches.

A combination of the two methods can be developed that has a computational and space cost proportional to the length of the sequence. Firstly, a chunk size is chosen. Then, the forwards recursion is calculated using the space-saving version of the recursions. The first matrix row encountered is then stored in a list. Each time a number of rows have been calculated that is a multiple of the chunk size, this is also stored in the list. This is done until the complete recursion has been calculated. The complete sub-matrix running from any stored row to the next (or the end of the sequence) can now be calculated as needed using the normal forward recursion, initialized on the stored row. The space-saving implementation of the backwards matrix can then be used to provide the backwards scores for each region, starting with the last and working towards the first, and counts can be added to the model trainer as normal.

This method requires the forwards matrix to be calculated twice, and also will need enough memory to store the forwards matrix rows for each of the chunks. However, this is significantly lower than the cost of storing the complete matrix. If the largest sequence that can be used for training with the current method is one megabase, then the chunk size can be set to once per half megabase. This would allow half a million chunks to be processed before using half of the available memory (the other half being required for calculating the sub-matrices). There are no sequences that we are aware of that are likely to exceed the order of a million, million nucleotides. Therefore, we propose that this method will allow single-head HMMs to be trained on

any practically available sequences without exceeding readily available memory resources. We plan to implement this training method in BioJava in the near future.

The investigations described in this chapter have demonstrated that the BioJava HMM APIs are highly adaptable to different model architectures, with potentially complex relationships between the values of parameters. The same implementation code was successfully used here for models with different numbers of states and for different emission alphabets. The results are numerically stable and fully probabilistic. These APIs have been used by others for modeling biological signals, for example, see (Hasan 2003). We hope that as the APIs mature, they will become used even more widely for different modeling tasks.

Chapter 4 Investigation of Recombination Rates Using SVMs

4.1 Introduction

In Chapter 3 attempts were made to divide chromosomes up into blocks with uniform but distinct properties using HMMs. The justification for this was the observation that certain biological processes appear to segregate with such patterns. Another way to look at chromosomes is to consider if there are associated properties that are continuous in nature. One property that appears to have this behaviour is the probability of recombination occurring between any two bases on the sequence.

Recombination events are responsible for the inheritance of a unique, mosaic combination of alleles during sexual reproduction. In humans it has become clear that the single nucleotide polymorphisms observed are grouped into regions bounded by points of recombination, which have recently been mapped for Chromosome 22 (Dawson, Abecasis et al. 2002). How much variation in recombination rates influences the inheritance pattern in organisms including man has to date not been quantified.

The exact mechanisms that drive differences in recombination rate are unknown. It has been shown in some organisms that some recombination hot spots can be directly controlled by very small regions of a chromosome, and that the trait of recombination rate is heritable (Dixon and Kowalczykowski 1991). It is always possible that the heritable component is something other than the genome sequence, such as the methylation pattern. The objective of this chapter is therefore to look for a sequence based signal.

Human Chromosome 22 provides the source of data for this investigation. The rate of recombination has been estimated along a large portion of the q-arm of human chromosome 22 between some 35 genetic markers (Dib, Faure et al. 1996; Dunham, Shimizu et al. 1999). With the advent of a finished sequence for this chromosome, it is possible to compare the genetic and physical distances. As indicated by the blue line in Figure 4-1, the recombination rate is not uniform across the region. By plotting physical position along the x-axis and the ratio of genetic to physical distance on the y-axis, the non-linearity shows up clearly as spikes. There may be recombination rate enhancing and repressing signals within the chromosome that are causing this position-dependant difference in recombination rate.

The process of learning how to predict recombination rate from sequence content can be addressed by a supervised learning approach. The dimensionality of the data is extremely high if we consider all possible sub-sequences within a region of interest. One methodology which has been used successfully for very high dimensional data is the support vector machine (SVM) which is now described in detail.

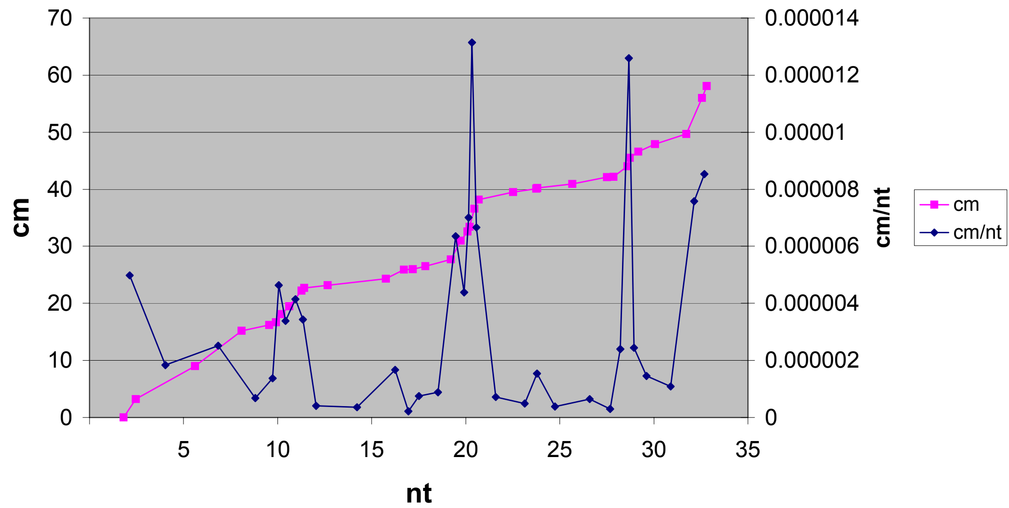

Figure 4-1 Comparison of physical and genetic distances along chromosome 22

The x-axis represents the genetic distance of each marker on the q-arm of chromosome 22 from the centromere, as measured in megabases. The pink line is the distance in centi-morgans of each marker from the first marker. The blue line displays a data-point between each consecutive pair of markers, with a height proportional to the ratio between the difference in the genetic distance between the two markers and their physical distance. Peaks in the blue graph indicate regions of relatively high recombination.

4.1.1 Support Vector Machines

Support vector machines (SVM) (Vapnik 1995) are linear models that use dot products and kernel functions to perform classification and regression tasks (see Section 1.3.1). They are designed to estimate an affine transform from an arbitrary dimensional input space to a single dimensional output space. This output space is defined as the distance of a data item from some plane in input space. The distance of a point from a plane can be computed as the dot-product between the point and the normal of the plane, plus the plane's constant (the smallest distance of the plane from the origin). All points lying within a plane satisfy the equation:

Equation 4-1 Equation of a Plane

$$v \cdot x - h = 0$$
$$v = \text{normal to the plane; } x = \text{any point in plane; h} = \text{plane offset}$$

Dot-products of sums can be re-written as sums of dot-products:

Equation 4-2 Normal to a Plane as a Weighted Sum of Vectors

$$\left(\sum_i x_i\right) \cdot x = \sum_i (x_i \cdot x)$$

It follows that we can represent the normal of the plane by a weighted sum of the training examples. Although most of the problems encountered do not have a good linear solution in the data-space, there is often a linear solution in some alternate 'feature space' that is equivalent to a non-linear solution in the data-space. For example, the space of all polynomial interactions of order two or less between the two components of a vector describes all conics in the data-space. If there is a kernel

function that computes dot-products in this feature space, Equation 4-2 shows that the equation of the plane can be implicitly represented as a weighted sum of the kernel functions acting upon each data point and the data-item x. The full equation becomes Equation 4-3 where β_i is the weight of the ith training example in the plane normal.

Equation 4-3 Definition of a Support Vector Machine

$$f(x) = \sum_i \beta_i k(x_i, x) - h$$

This formulation has the interesting property that although the solution is in a potentially large feature space, there is exactly one more parameters than there are training examples. The standard SVM training algorithms introduce a further constraint upon the parameters such that one of them is not free. This means that regardless of the resulting model, it can contain no more information than the original training data.

When using a SVM, it is usual to vary x for some fixed range of values for x_i and β_i. This can be made more explicit by rewriting the kernel function term as a basis function:

Equation 4-4 Basis Functions for Kernel Functions and Data Points

$$\phi_i(x) = k(x_i, x)$$

This allows the SVM equation to be re-written as follows.

Equation 4-5 SVMs in Terms of Basis Functions

$$f(x) = \sum_i \beta_i \phi_i(x) - h$$

If any of the weights β_i are zero, then the associated basis function does not affect the result. If only a few of the weights are non-zero then the resulting function is relatively simple. This property is called sparsity and the data-points associated with the contributing basis functions are called support vectors, as for separable problems, when displayed graphically in the feature-space (and if the feature space is related by a continuous function to the data-space, in that also), they 'support' the learned hyperplane (or decision boundary in the data-space). For the special case of pair-wise classification, the support vectors are the data points that lie on the boundaries of the convex hulls of the two sets of data in feature space.

There are several methods available to estimate the parameters of the hyperplane (β, h) given an error function. We found the most efficient method currently available to be the SMO algorithm (Platt 1998). This method optimises for pairs of support vectors at a time, and eventually this leads to the complete solution being found. For problems with appreciable sparsity, this method takes time approximately proportional to the training set size. Once trained, SVMs are computationally very cheap to apply to new data items, assuming that the kernel function is easy to compute.

4.1.2 BioJava APIs for Support Vector Machines

Support for SVMs is provided by BioJava. Their implementation builds upon the formulism discussed in Section 1.3.1 by providing a Java interface called `KernelFunction` that computes the kernel function for any two Java objects. There are now a number of other publicly available SVM implementations, such as SVM-

fu[32]. However, unlike many of the other implementations, BioJava allows arbitrary kernel functions to be used. One of the benefits of this flexibility is that we can manipulate the data inside the kernels, for example normalizing vectors onto a unit sphere or scaling some subspace. We have also observed that for complex kernel functions, the performance of the BioJava implementation does not degrade.

4.2 Methods

4.2.1 Searching for a Signal Affecting Recombination Rates Using a Word-Frequency Kernel Function

Under the assumption that there are sequences within chromosomal DNA that affect recombination rate, and given example sequences known to have high or low rates, it should be possible to discover some metric of the sequence which is predictive of its recombination rate. It is possible that recombination rate is mediated by very simple sequences (e.g. poly-A or poly-GC), or relies upon very complex patterns (e.g. entire promoters or histone-binding regions). Either way, it is likely that part of the signal will correlate with scores collected by counting word frequencies (such as the frequencies of all octamers).

Once the sequence data is transformed into a format that is equivalent to counts over a finite set of properties, it becomes suitable data for processing with a Support Vector Machine using a simple kernel function. The transform from sequence to counts can be considered to be equivalent to a data-to-feature space transform, inducing a new kernel function that both projects sequences to counts and then calculates the dot-product of the counts.

---

[32] SVM-fu is distributed through the http://www.ai.mit.edu/projects/cbcl/ web site

In previous studies with word-count based kernel functions, greater accuracy has been achieved by normalizing the counts prior to calculating the kernel function as it is usually the relative proportion of the different words and not the absolute count that is informative, and normalization both controls for document size and numerical instabilities introduced by large differences in size between the magnitudes of training data (for example, see example training data accompanying svm-light[33]). However, this normalization can actually be performed within the kernel itself (Equation 4-6) as the process of normalizing each input vector is itself a transform from some data-space to a feature space (projection of all points onto a unit hyper-sphere).

In the cases when the data-space is very large and the cost of normalizing this is prohibitive but the un-normalized kernel is cheap to compute, the normalizing kernel saves both space and time. It potentially makes reading the computer code easier as the entire data-to-feature transform is represented in one place, the kernel, rather than being spread amongst multiple pre-processing steps. If repeatedly calculating the terms $\langle a,a \rangle$ and $\langle b,b \rangle$ is found to be expensive then these values can be cached. By applying an object-oriented design methodology, we can implement a kernel that delegates to an underlying kernel function for all values not known and caches the results for all terms of the form $x \cdot x$ for quick access. This is the approach taken in the BioJava toolkit, and we have found that it drastically decrease the computational load of kernel functions such as the normalizing and radial basis kernels that require some values to be repeatedly calculated.

---

[33] See http://svmlight.joachims.org/ for an example of using normalised counts for classification

Equation 4-6 The Normalizing Kernel

$$k_{\langle\cdot,\cdot\rangle}^{norm}(a,b) = \langle \hat{a}, \hat{b} \rangle = \frac{\langle a,b \rangle}{\sqrt{\langle a,a \rangle \langle b,b \rangle}}$$

The family of kernel functions used in this study can be represented as:

Equation 4-7 SuffixTree Kernel

$k_{k_{df}^{SuffixTree}}^{norm}(a,b)$ where

$$k_{df(\cdot)}^{SuffixTree}(a,b) = \sum_{d \in (0..l)} df(d) \cdot \sum_{i \in \Omega^d} a_i \cdot b_i;$$

df = depth function (scales tree counts by depth);

suffix trees are indexable by string i

Because of how the suffix trees are constructed, they will not contain nodes for zero counts. By definition, if a given sub-string is absent from the entire sequence, then all other sub-strings containing that string will also be absent. For this reason, the nodes of the tree are sparsely populated (only nodes that store non-zero counts are instantiated). Thus, the index i in Equation 4-7 need actually only loop over values of Ω^d that are populated in both a and b.

The depth function term $df(\cdot)$ allows the counts associated with strings of a particular length to be given greater or lesser weight. For example, a depth function that always returns 1 will leave the counts un-scaled (uniform depth function). A depth function that returns non-zero for one value and zero for all others will have the effect of only including words of a single length. A depth function of the form $|\Omega^d|$ will make longer matches more significant than shorter ones, taking into account the fact that they are less likely by chance (normalizing depth function).

In principal, the depth functions are a subset of functions that return a scale factor for each i that is purely a function of its length. In cases when the sequence bias is significantly divergent from uniform, it may be worth re-defining Equation 4-7 in terms of a per-i scaling function instead of a depth function. However, we considered that in this case any increased accuracy obtained would be off-set by the need to optimize the scaling function, and any associated computational overhead.

4.2.2 Construction and Training of an SVM for Predicting Recombination Rate

The SVM used was constructed with the normalized suffix-tree kernel as described above. Both the uniform and normalizing depth functions were evaluated. The maximum tree depths were fixed from 1 to 9 for the uniform model and 1 to 8 for the normalized model. The models were trained using the SMO method for classifiers as at the time the BioJava implementation of SVM regression that was not numerically stable.

All clones in the partially finished Human chromosome 22 were extracted, together with their approximate coordinates within the chromosome. These were fully repeat-masked for both simple and complex repeats using repeat masker[34]. The chromosomal locations of all 35 markers were used. The high-recombination rate region within approximately 18-21Mb and the low recombination rate region within approximately 21-17Mb were used as the positive and negative training sets respectively. All clones from these regions were used for training. All clones outside of this region were included during the prediction phase.

---

[34] se here for an online reference for repeatmasker, currently unpublished:

http://ftp.genome.washington.edu/RM/RepeatMasker.html

4.3 Results

4.3.1 Recombination Rates Predictions

In no cases could any of the models with a depth of less than four be trained. This indicates that the information necessary to predict recombination rates relies upon sequences of at least four in length. To bin sequences into two categories should only have required one variable, or the ratio of two variables, so the tree depth of 1 or 2 should have been sufficient to trivially separate them if the signal was purely based upon low-order sequence bias (such as AT/GC ratio).

For the models with maximum depth 4 and upward, the SVM produced an output centred on 0.0, indicating that the model is not consistently predicting items as being positive or negative. During training, the procedure attempts to predict a function such that all negative examples have a value less than –1.0, and all of the positive examples have a value of greater than +1.0. All items that are within the range –1.0 and +1.0 will be support vectors. If unseen data has an output between –1.0 and +1.0, this indicates that the SVM considers this data-point ambiguous, but it still attempts a classification that can be read by looking at the sign of the output. Values of magnitude larger than 1.0 indicate that the SVM was confident in its assignment.

The models trained using the uniform counts (Figure 4-2, Figure 4-3, Figure 4-4) learned functions that model the training data well, giving outputs around +1 for the high recombination regions and -1 for the low recombination regions. The higher depth models (6-9) produce SVM outputs that are noticeably closer to zero and less spread than the lower depth models. However, the general shapes of these curves (as judged by the 10 point moving average) are very similar. Arguably, the output shadows the recombination curve, particularly around the peak at 10Mb, and the dip

at 30Mb. However, it also predicts a recombination-poor region at 5Mb. The 5Mb feature may be biologically significant but not visible in the recombination plot due to marker density, or may be an artefact.

The models trained using length-normalized counts, Figure 4-5 Figure 4-6, Figure 4-7) are less prone to wild fluctuations (compare the scattering visible in Figure 4-2 and Figure 4-5), which may indicate that the predictions are more robust. In addition, the dynamics of the predicted recombination frequencies are more similar to the actual rates (Figure 4-6, Figure 4-7). Again, as the depth of the suffix-tree is increased, the resulting function becomes smoother, closer to zero and fluctuates less wildly.

4.3.2 Cross-Validation

To assess how robust the predictions of the SVMs are with respect to the training data, the sequences within the positive and negative training sets were partitioned into three sets randomly. Three models were trained, one for each partition, and then tested by predicting the membership of the other two partitions. This 3-way jack-knifing was performed for all depths in the normalized models using training methods which where otherwise identical to those used above. The best accuracy of 80 % (random 50 %) is achieved for a depth of 5, with accuracy becoming worse for greater depths.

The resulting models appeared to be memorizations of the training data, with very few example sequences not included as support vectors (between 1 and 4 training examples left out). They do seem to be consistent among one another (depth of 5 being the most reproducible), with prediction accuracies that are consistent, both inside the training data (Figure 4-8), and across the entire chromosome (Figure 4-9).

However, over all, the chromosomal predictions are less informative of recombination rate. This is as to be expected with less training data.

Error! Not a valid link.

Figure 4-2 Total Results of Training the SVM using Uniform Counts

SVM predictions are displayed as points on the scatter graph. For comparison, the recombination rate is also displayed.

Error! Not a valid link.

Figure 4-3 Moving Average for Uniform Counts models of Depth 4-6

10 point moving averages of the SVM predictions are displayed for models with depths of four, five and six. For comparison, the rate of recombination is also displayed.

Error! Not a valid link.

Figure 4-4 Moving Average for Uniform Counts models of Depth 7-9

10 point moving averages of the SVM predictions are displayed for models with depths of seven, eight and nine. For comparison, the rate of recombination is also displayed.

Test High Low Test

Test High Low Test

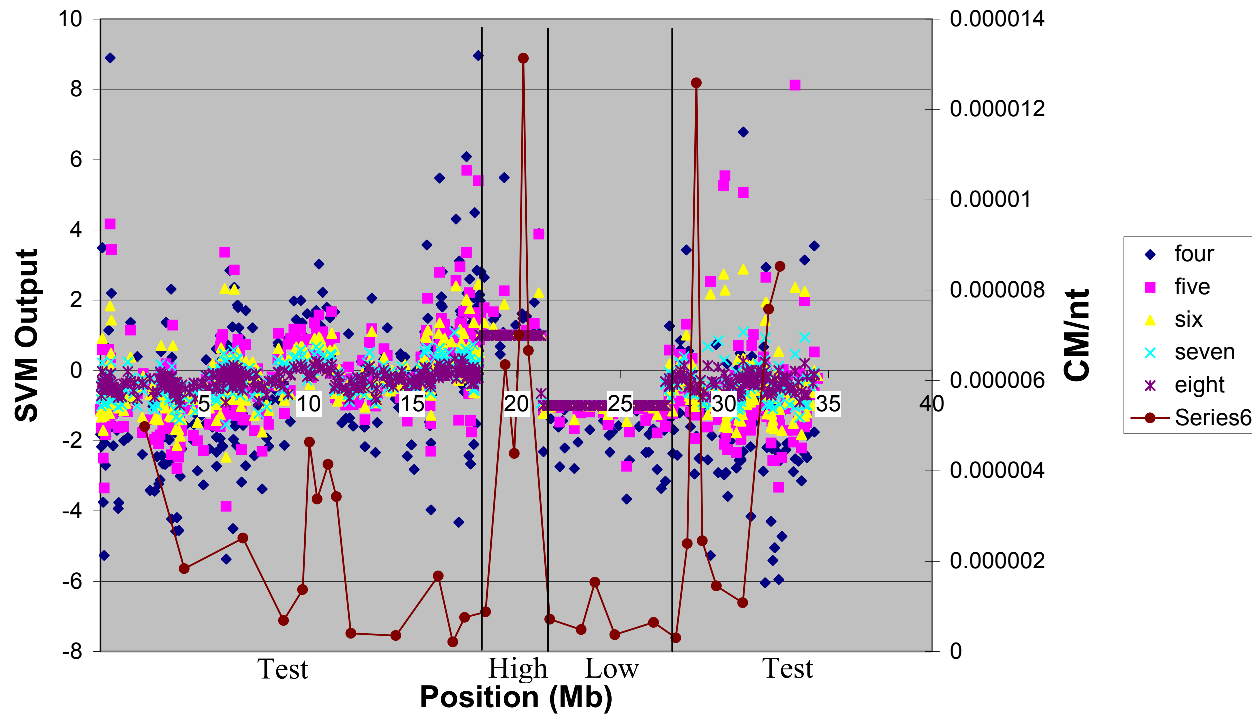

Figure 4-5 Total Results of Training the SVM using Normalized Rates

SVM predictions are displayed as points on the scatter graph. For comparison, the recombination rate is also displayed.

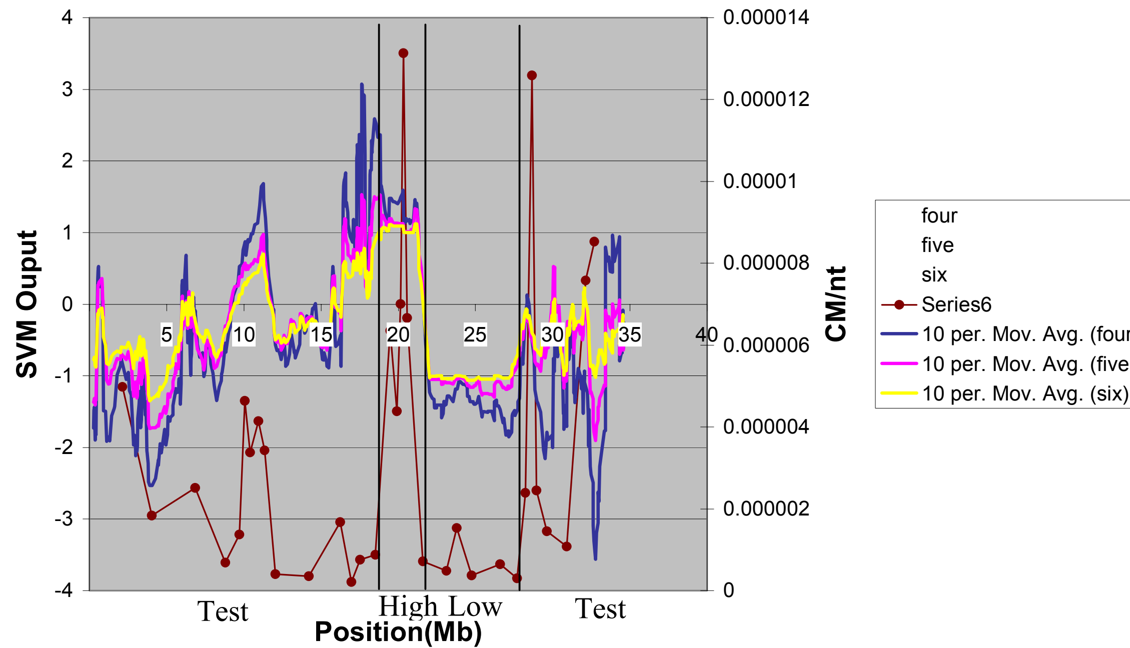

Figure 4-6 Moving Average for Normalized Rates: Depths 4-6

10 point moving averages of the SVM predictions are displayed for models with depths of four, five and six. For comparison, the rate of recombination is also displayed.

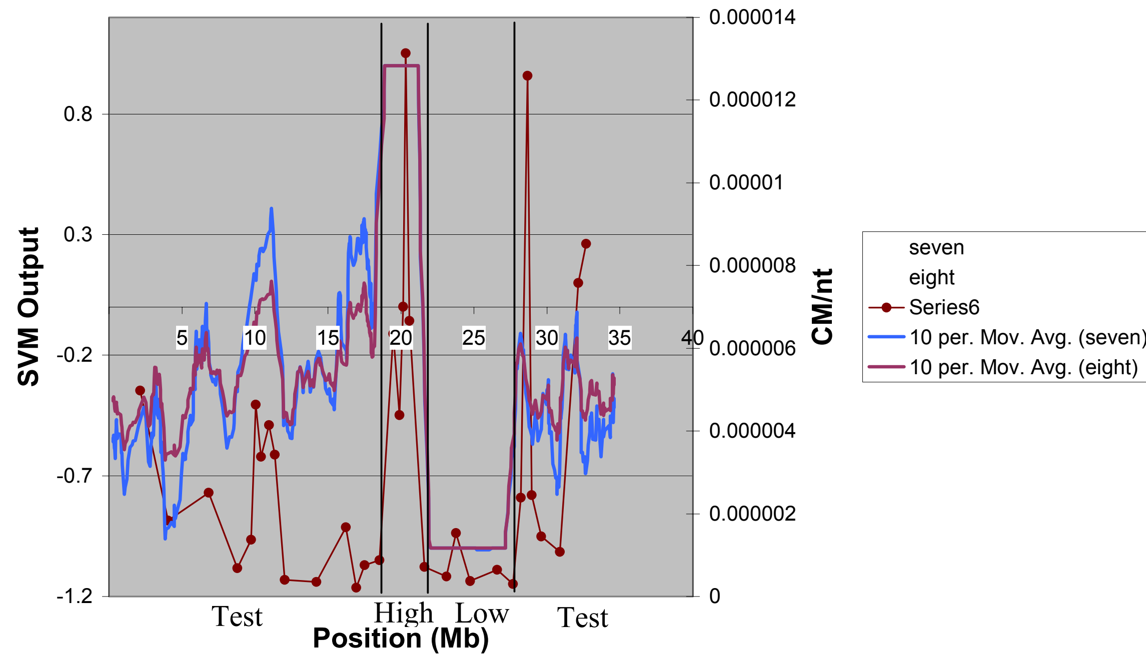

Figure 4-7 Moving Average for Normalized Rates: Depths 7-9

10 point moving averages of the SVM predictions are displayed for models with depths of seven, eight and nine. For comparison, the rate of recombination is also displayed.

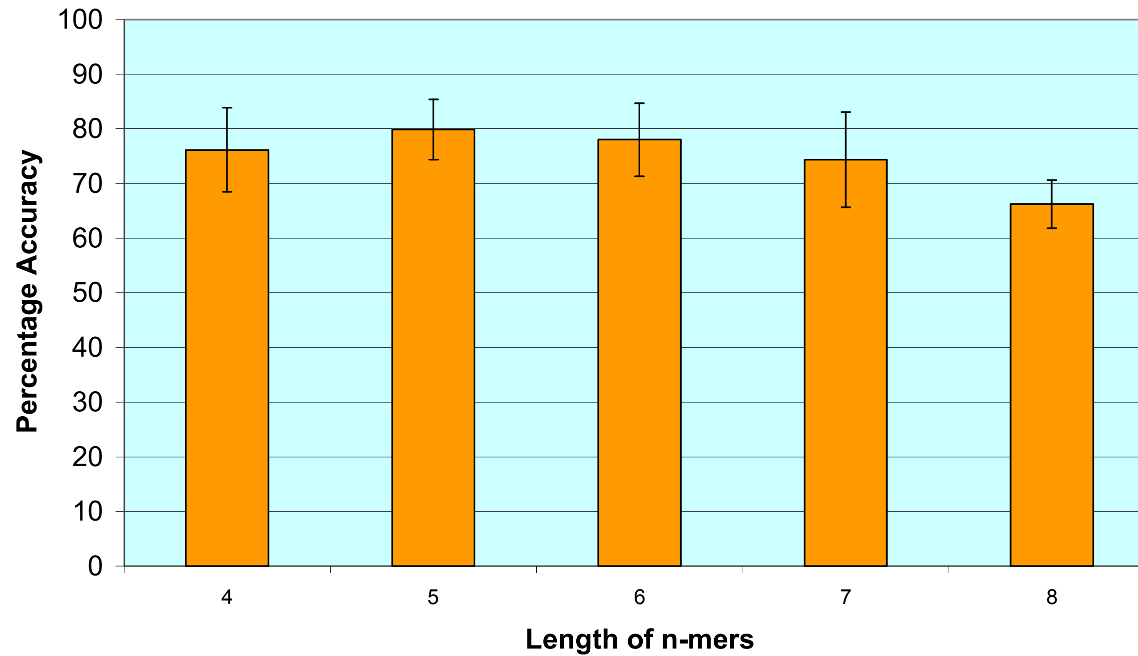

Figure 4-8 Accuracy for Recombination SVMs Under 3-Way Jack-knifing

The percentage accuracy for each group of jack-knifed models is displayed as a bar. The y-axis displays the percentage accuracy of each group of jack-knifed models. Error bars represent two standard deviations. The bar representing 5-mers has the greatest height.

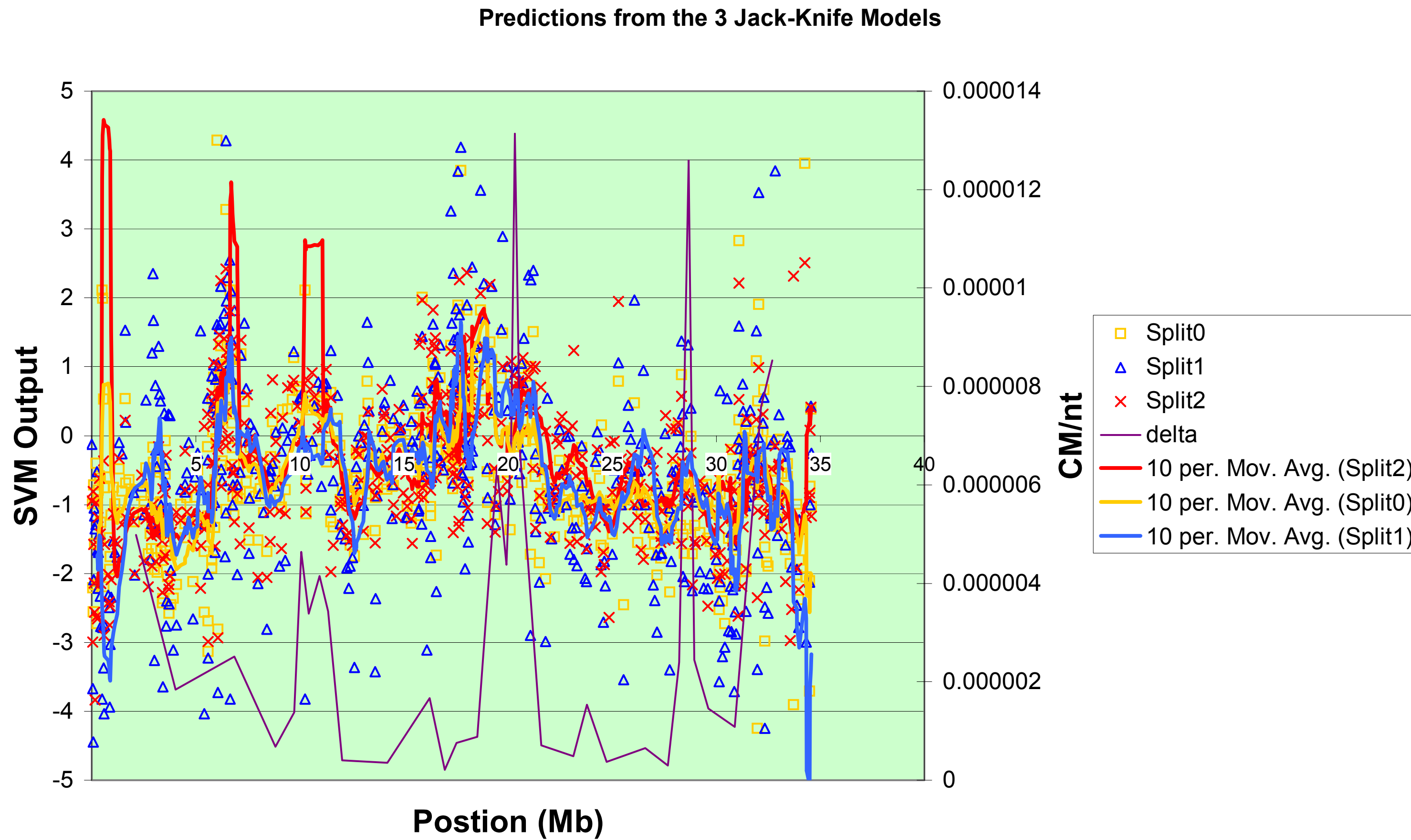

Figure 4-9 Predictions Across the Entire Chromosome from the 3 Jack-knife Models for Depth of 5

4.4 Discussion

It is clear that the normalized rates kernel appears to be more robustly predicting recombination rates than the uniform counts kernel. This exposes one of the interesting properties of SVMs when dealing with data containing errors. The relative magnitude of each data-space basis (here each word) acts in the spirit of a prior over how likely this basis is to form part of the normal to the hyper-plane. In theory, the training method for SVMs should allow these affects to be removed. In practice, with incomplete and stochastically sampled data sets it seems to be important to try to make sure that the raw data is presented in such a way as to normalize out any biases (such as normalizing vectors, pre-normalizing each dimension and the like). It is possible that the RVM methodology (explored in Chapter 5 for a different type of problem) would be more robust for this kind of data as during training it makes an estimate of the degree of certainty with which parameters can be set. This would have the effect of removing dimensions that sharply but inconsistently affect the predicted value regardless of their magnitude. SVMs will tend to be locked into local minima associated with these sharply varying dimensions, as they are greedy algorithms (they search for a global maximum by hill climbing).

The jack-knife results indicate that the signal being learned is not strongly dependant on the training data. This is evidence that these models are learning real signals. It is interesting that the 3 jack-knife models appear to be memorizing their training sets but still generalize in comparable ways to each other. Because the function learned is represented as a sum of normals to a plane (each support vector representing one of these normals), different combinations of support vectors and associated weights may be constructing a very similar hyper-plane. It would be interesting to attempt to calculate the total normal to the hyper-plane in each case and

in the case where all training data is used to find the short sequences that are actually informative. This has not yet been attempted, as the raw data has gone though several data-to-feature space projections before arriving in the space for which the hyper-plane is constructed and it is not clear how to represent these transforms in terms of raw word counts.

Recombination hot spots are still not, to our knowledge, fully explained or predictable by any method available. However, recently there has been evidence that poly GT/AC tracts may contribute to differences in recombination rates (Gendrel, Boulet et al. 2000; Majewski and Ott 2000). In the light of this, it would be worth re-running the analysis with the now finished and un-masked chromosome 22 sequence. The masked sequences used here almost certainly had most of this signal screened out. An important message to this story is that 'junk' DNA may well have functionality within the genome, and should not be discarded out-of-hand. The genome contains sequences with many functions, many of them which are not directly related to coding for valid proteins.

Chapter 5 RVMs for Classification of Expression Data

5.1 Introduction

The phenotypic behaviour of a cell is in large part due to the activity of its proteins. These are translated from mRNAs, which are have been transcribed from active genes. There are many levels at which the activity of proteins can be regulated, however it has generally become accepted that measuring mRNA levels gives a good insight into the relative levels of gene activity. The results of simultaneous measurements of large numbers of mRNA levels, made in a single experiment, will be referred to as 'expression data'.

It has become possible to collect expression data systematically using methods such as quantitative PCR (Buck, Harris et al. 1991; Nedelman, Heagerty et al. 1992), array technologies (Schena, Shalon et al. 1995; Lashkari, DeRisi et al. 1997; Shalon 1998) and DNA chips (Guo, Guilfoyle et al. 1994; Hughes, Mao et al. 2001). The availability of complete genomes and fairly complete gene annotation has enabled the construction of DNA probes to represent most expressed mRNAs in a particular population of cells. Many thousands of these probes can be arrayed onto a single microarray or DNA chip, making it possible to capture a snapshot of the expression state of a cell in a single measurement.

Given the availability of this measurement technology, it becomes possible to take snapshots of a range of conditions of a population of cells and, by processing the results, compare the expression levels of different genes under different conditions. Examples include comparing the natural state of the cell with its state during

conditions of metabolic stress, heat shock or disease. These measurements generate large volumes of data that can contain large statistical errors as a result of limitations inherent to the experimental technologies. Errors may also arise from stochastic variations in the different cell populations under study, due to the inherent dynamics of complex systems (for example, see (Guillouzic, L'Heureux et al. 2000; Smolen, Baxter et al. 2001) for discussions of ways in which the dynamics of gene expression are inherently complex systems).

For many cellular processes, we have a fair understanding of the ways groups of genes are co-regulated as a result of biochemical, genetic and other analysis. Expression data gives us the opportunity to systematically extend this understanding to the whole genome, showing previously unknown regulatory relationships. The expectation is that genes which appear to be co-regulated are likely to be involved in the same cellular processes. One way of viewing this data is from the point of view of genes, for example, the level of a gene during sporulation (Chu, DeRisi et al. 1998). Another way is to classify conditions, i.e. to match a particular cellular expression snapshot to a particular cancer condition (Alizadeh, Eisen et al. 2000; Ramaswamy, Tamayo et al. 2001).

The standard method for processing expression data is currently cluster analysis (Eisen, Spellman et al. 1998). This describes the dynamics of expression data as a hierarchical model, either in terms of similar experimental samples given genes, or similar genes given a set of experiments. Many of the major signals that emerge from naïve clustering are strongly correlated with histological features, for example in different areas of the gut (Bates, Erwin et al. 2002), or different developmental stages (Mody, Cao et al. 2001). Sometimes, by selecting sub-trees of genes, sets can be

found that co-segregate with a qualitative or quantitative observation. However, this is far from being an automated task, relying on human concepts of relevance and relatedness.

One practical application of expression data analysis using machine learning techniques has been the classification of cancer cell types from different patients. Different cancer cell types can appear very similar, but may have very different survival rates which may require different treatments (Kihara, Tsunoda et al. 2001; Liefers and Tollenaar 2002; van 't Veer, Dai et al. 2002). In (Ramaswamy, Tamayo et al. 2001) SVMs were used to construct a classifier for the expression patterns of 14 distinct tumour types from 218 samples. Each expression data sample included measurements from 16,063 genes. The SVM-based classifier could then be used to classify the tumour type of any new sample with a high degree of accuracy (78%). They were also able to identify which genes contributed most to the SVM model.

SVMs were able to carry out the above classification as a result of the large expression differences between cancer cell types. A much more difficult problem is to automatically extract information about changes in gene expression as a result of a much subtler difference, such as in response to drug treatment. The subtler effects tend to be masked by the differences between the cell lineages that have been treated.

Perou (Perou, Sorlie et al. 2000) describes a series of experiments measuring 8,102 gene expression levels in human breast cancer cell lines and biopsy samples. One sub-set of the samples are biopsies taken from patients with tumours before and after treatment with the anti-cancer drug, doxorubicin. The expression data for 20 before-after pairs were clustered using hierarchal clustering. All but three of the sample pairs clustered together as siblings in the tree. This indicates that the changes in gene

expression due to treatment are not reliably detected by clustering against the background of the cell lineage. In a further paper (Brenton, Aparicio et al. 2001), cluster-analysis could be used to identify sub-types of breast tumours. However, this required much larger amounts of data and some human intervention. It also was unable to address the issue of what effect the doxorubicin had upon gene expression.

Typically, when modelling expression data, the aim is both to perform some classification task, and also to look at the resulting model and identify genes that contribute to the model, with the hope that they will provide biological insight. In this chapter the use of SVMs and RVMs is explored to extract information about the effects of doxorubicin. We both evaluate whether these methods are able to generate models that can classify micro-arrays into pre- and post-treatment with doxorubicin, and also to decide if the models they produce are consistent with the known biological processes, and therefore could be used in other situations to identify novel relevant genes.

5.2 Cellular Responses to Doxorubicin

Doxorubicin causes cellular apoptosis by several routes. The primary action of doxorubicin activity is due to intercalation into double-stranded DNA. This both prevents the normal UV-repair pathway (nucleotide excision repair) and causes single-strand breaks, both of which lead to an increase in the rate of DNA repair enzyme activation. Normally, topoisomerase II relaxes tension due to supercoiling by scanning DNA. It then breaks one of the two phosphate backbones, allows the DNA to relax and then repairs the resulting single strand break. If instead it binds to an intercalated doxorubicin molecule, the single-strand break is made, but is not repaired. In this case, the topoisomerase II protein remains covalently attached to the broken

strand (Tewey, Rowe et al. 1984). The activity level of the DNA repair response is measured by the cell, and if it increases above a critical threshold, the cell enters an apoptotic response.

Cell lines that are resistant to doxorubicin often share a common set of mutations. Topoisomerase II activity can be severely impaired (Potmesil, Hsiang et al. 1988). This is consistent with the role of this gene in the drug's mechanism of action. Resistant cell lines frequently express multi-drug resistance proteins, such as P-glycoprotein (P-gp), and multi-drug resistance associated protein (MRP) (Grandjean, Bremaud et al. 2001) which expel the drug from the cells. Impaired systems for maintaining levels of small ions, such as Na^+ and K^+ (Lawrence 1988; Lawrence and Davis 1990) also seem to confer a measure of resistance. It is possible that these small ions are required to enhance the stability of the complex between topoisomerase II and the DNA. Resistant cells often have impaired Jun-Fos pathways (Pourquier, Montaudon et al. 1998). During apoptosis, the Jun-Fos transcription factor heterodimer is activated via a signal-cascade. This in turn leads to the altered expression of gene products, activating the signal-cascades mediating the cellular apoptotic pathway. If either Jun or Fos gene is mutated to loss-of-function mutants, then is apoptosis pathway can not activate.

5.3 Generalized Linear Models

Although SVMs have been used successfully to distinguish between tumour types where there have been large numbers of samples available as described above (Ramaswamy, Tamayo et al. 2001), previous implementations guarantee that the SVM will find a solution even if there is insufficient evidence to support it. The training algorithms for SVMs search for the globally 'best' separating hyper-plane,

and give no indication of the range of hyper-planes that perform similarly well, even if they have a radically different plane normal. This brings into question their utility for discovering new expression relationships in small or noisy data-sets, as it becomes difficult to distinguish between results that are significant, where all 'good' hyper-planes have very similar normal vectors, and those that are correspond to the 'best' but uninformative solution, where a wide range of normal vectors would perform nearly as well. In this chapter, we apply a Bayesian approach to training that is able to address this.

In Section 4.1.1, we discussed how SVMs can be represented as a sum of basis functions (Equation 4-5). The general class of models that take on this form are called Generalized Linear Models (GLMs) (Nelder and McCulagh 1983). During training, the selection of the weights is just a scaling factor for each subspace, stretching the dimensions that increase the accuracy of the model, and shrinking those that are irrelevant. From this point of view, the basis functions each define a dimension in the feature space under consideration.

Some of the basis functions will be highly correlated with one another. This means that using more than one of these will contribute little or no additional information. Other basis functions may simply be uninformative to the problem in hand. By defining some measure of the information contributed to the model by a given basis function, and the additional complexity of including that function, it is possible to make a trade-off between the simplicity of a model and how well it fits the data.

Bayes Theorem states how the probability of simultaneously observing two events is related to the probability of observing one event in isolation and the probability of

observing the second event given that we already know that the first one has occurred. Let us consider the case of observing a model, m and data, d.

Equation 5-1 Bayes Theorem

$$\begin{aligned} p(m,d) &= p(m \mid d)\,p(d) \\ &= p(d \mid m)\,p(m) \end{aligned}$$

This equivalence can be re-arranged to express one of the conditional probabilities in terms of the independent probabilities and the other conditional probability. It is in this form that Bayes Theorem is most often presented.

Equation 5-2 Rearrangement of Bayes Theorem

$$p(m \mid d) = \frac{p(d \mid m)\,p(m)}{p(d)}$$

The terms in this form all have names in Bayesian statistical analysis.

Equation 5-3 Bayes Theorem in Words

$$posterior = \frac{likelihood \cdot prior}{evidence}$$

In the case of models and data, the posterior is the probability of our model (and associated parameters) given the data. The likelihood is the probability of observing the data given our model. The prior is the degree of belief we have that the model is sensible. The evidence is the probability of observing our data given any possible model, which in practice means the sum or integral of the probability of observing the data over all possible values for all parameters of the model.

One method for training GLMs which makes use of Bayesian statistics is the Relevance Vector Machine (RVM) (Bishop and E. 2000; Tipping 2000). In the case

of RVMs, the prior is chosen in such a way that it favours models where many of the weight parameters have values near to zero. For a particular parameter set to have a high posterior probability, the prior "cost" of any non-zero weights must be balanced by an increased value of the likelihood. If a particular basis function does not contribute to the likelihood sufficiently, then a greater overall posterior can be achieved by setting its weight to zero. RVMs can be trained by selecting parameters that maximize the posterior (Tipping 2000), or by fitting a variational approximating distribution to the posterior (Bishop and E. 2000).

A pure Java implementation of the RVM method has been implemented (Down 2003). This implementation uses patterns similar to the BioJava SVM implementation (Section 4.1.2) to insulate the optimiser from the data. An interface `BasisFunction` is provided that has a single method that returns the value of the basis function for a Java object. There is also an interface `BasisSource` that represents an iterater over a set of basis functions. The known implementations of RVMs (Bishop and E. 2000; Tipping 2000; Down 2003) all have space and time costs that scale very badly with the number of basis functions being considered. The API for Down's method employs the 'small working set' heuristic to work around this. During training, many weights become sufficiently close to zero to be discarded within a very few cycles of optimisation. This is exploited by setting a high and low water-mark for the set of basis symbols being considered. Initially, basis symbols are obtained from the `BasisSource` until the high water-mark is reached. The optimiser then runs until it has discarded enough basis functions that the low water-mark is reached. At that point, basis functions are added until the high water-mark is again reached, all parameters are re-initialised and the optimisation is resumed. This process is continued until the `BasisSource` has no more basis functions available. At this point,

the optimiser runs until the model converges. This heuristic keeps the cost of training a model with increasing numbers of basis functions proportional to the total number of basis functions that must be considered, and some function of the working set size. In practice, this makes some problems tractable that would be otherwise intractable.

The RVM API of Down's implementation interacts with the BioJava APIs for SVMs. Where appropriate, interfaces for representing training data and models are reused. In addition, there is adaptor code that allows a kernel function and a set of training objects to be viewed as a `BasisSource` over the implied basis functions (see Equation 4-4). In practice, very few lines of code need be changed to switch between analysing a data set with SVM and RVM methods.

5.4 Micro-array Classification Using a Support Vector Machine Implemented as a Linear Kernel RVM

To investigate the behaviour of SVMs when applied to a hard expression analysis problem, we applied them to the dataset described above (Perou, Sorlie et al. 2000).

The BioJava implementation of SVMs was used to construct a classifying support vector machine using the dot product (linear) kernel function to evaluate expression data. The kernel function was implemented so that the expression data was represented as an array of the log of the ratio between background and experimental sample levels. This was trained using the complete set of expression data described in Section 5.1 using the SMO training algorithm. The resulting model contained nearly all micro-arrays as support vectors. This suggests that the model was effectively memorizing the training set. We therefore decided not to further investigate the use of classically trained SVMs for this task, as they seem to be unable to model this problem.

To investigate whether the SVMs were extracting any significant data from the expression data-set, or just memorizing it as suspected, we applied an RVM approach (Section 5.3). An RVM was constructed with a `BasisSource` using the above training data and kernel function to generate basis functions (See Section 4.1.1, and in particular Equation 4-4 and Equation 4-5). The RVM was then trained using the complete set of micro-arrays. Given these basis functions, the RVM becomes equivalent to a Bayesian interpretation of the SVM. This RVM rejected all basis functions during training. This indicates that none of the SVM solutions using a linear kernel function robustly describes how to separate the pre- and post-treatment samples.

This negative result does not necessarily mean that this task could not be performed with either an SVM or an RVM using linear kernel functions, but that there was insufficient training data to support any parameters. By working with larger training sets, or more complex kernels, it may be possible to apply a kernel RVM to this data. However, this result does indicate one of the main benefits of RVM training over SVMs in that the RVM was able to indicate that no reasonable model could be produced. The SVM produced the best model that it could, which was of poor quality, but without any indication to the user that this was the case. Any predictions made on the basis of genes contributing to the separating hyper-plane are likely to have been incorrect, but there would have been no way to know this purely from the SVM results themselves.

5.4.1 Framework for Generalised-Linear-Models amenable to Expression Arrays

Given the failure of the linear kernel model description used above to discover expression differences resulting from treatment using doxorubicin, we now present an alternative way to model the problem.

An individual array measurement can be considered as a tuple of measurements with one dimension per spot on the array. This is a convenient interpretation for database storage and cluster analysis. Another point of view considers each spot to be the result of evaluating a probabilistic function on the particular sample (the log of the ratio of measured expression levels in experimental and background samples). This interpretation takes into account that the expression level measured is subject to noise. It transforms individual measurements (and by extension the individual genes) into entities amenable to hypothesis-directed reasoning using the RVM framework (Section 4.1.1), as now each measurement for each gene can be treated as the value of a basis function.

Consider a set of genes, G, a set of micro-arrays, A, and the function that retrieves the level for a gene on an array, $l(g \in G, a \in A)$. For any particular fixed g, there exists a conditioned version of this function, which we shall call $l^g(a)$. A GLM can then be constructed where the set of functions being evaluated is the set $l^g(.)$ for each gene. This model produces an output based upon a weighted sum of the log ratios of expression levels of multiple genes that is potentially predictive of some process.

Given any pre-defined classes by which the array measurements can be classified, a GLM can be estimated to perform that classification. If the sparse training approach is taken, then the hope is that the model will tend to extract key genes that have a type of

response that helps in the classification task, and will tend to discard all uninformative genes. This has the beneficial property of giving back a list of genes that are representative of each distinct response to the stimulus that aids in the classification task. If multiple genes share the same or similar expression profiles, the sparsity properties of the trainer will tend to find the statistically most representative member of that group and discard all others. For some uses of the method, such as where a complete list of significant genes would be useful (including those contributing similar information), this property is a disadvantage. In these cases, some further analysis of the data will be required to recover these other genes from the training data.

5.4.2 RVM Analysis Using the Small Working Set Heuristic

To evaluate this approach, a training set was constructed containing all of the before and after treatment measurements introduced above. The aim was to classify micro-arrays into those before and after doxorubicin treatment. An output of 1.0 would indicate that the method was certain that it was an example of ‘before’. An output of 0.0 would indicate that the method was certain that it was an example of ‘after’. A value between these two values indicates the degree of confidence that the sample belongs to one class or the other.

The number of basis functions to be evaluated was very large (one for each of the 8,102 genes). It was not practical to train the RVM with all of these simultaneously, so the small working set heuristic, described above (Section 5.3), was employed. The high water-mark was set to 90, and the low water-mark was set to 75. As long as the total number of basis functions needed for the task is below the low water mark, we could expect the result to be unaffected.

To check whether the heuristic altered the result the training was performed three times, using different permutations of the training data and of the order that the functions were added. The three models produced were identical (the same genes with weights within the bounds of numerical precision), and gave the model shown in Table 5-1. This suggests that, with this data set, the small working set heuristic works.

Table 5-1 GLM for all before-after pairs (to 4 s.f.)

Accession	Weight	Gene Name	Description
AA017544	-3.269	RGS1	Regulator of G-protein signalling 1
T72398	4.982	TDO2	Tryptophan 2,3-dioxygenase
AA040944	-6.299	FOS	Transcription factor involved in the apoptotic pathway

This model correctly classifies all of the training examples using the log-ratios of just three genes. Of course, training and testing on the same data-set is not robust for assessing how well models generalise, but the simplicity of the model suggests that this approach may work. Additionally, one of the three genes used is FOS, which is known to be involved in the apoptotic pathway activated in response to doxorubicin treatment (see Section 5.2).

To assess how reproducible these results where, we performed a "leave one out" cross-validation. For each of the forty micro-arrays, a prediction was made using a classifier trained on the remaining thirty-nine micro-arrays. The accuracy rate of the model for unseen data can then be estimated as the average accuracy of these forty predictions.

Of the 40 different models generated, 29 predicted the unseen item correctly. This is an accuracy rate of 72.5%, compared to the expected rate of 50%. All of the correct predictions typically had extreme probabilities (< 0.2 or > 0.8) where as the incorrect predictions were all relatively close to 0.5 (> 0.3 and < 0.7). 15 models used three genes, 23 used four genes and 2 used five genes. Across these models, a total of 22 different genes were used. Every model contained AA040944 (FOS). 22 of them used AA027832 (HBA2) and 17 used AA017544 (RGS1). These results are summarised in Table 5-2.

In the forty models generated by cross-validation, several of them use one of two alternative probes for the gene TOP2A. The degree of reproducibility or otherwise of the levels associated with those two probes can be taken as an indication of the quality of the data-set. Figure 5-1 shows a scatter plot with one data point for each of the 40 micro-arrays, and x, y co-ordinates given by the level of expression for the two TOP2A probes in a given micro-array. The levels have an R^2 value of 0.68, indicating that although they are correlated, there is a considerable degree of independent variation.

A summary of the expression data for these probes is displayed in Figure 5-2. As is seen from the graph, none of the probes used in the models have clearly separate distributions before and after treatment. FOS, which is used by every model generated during cross-validation, shows differences between the two groups, as does JUN, and to a lesser extent, both of the TOP2A probes. However, it should be clear from this that there is no one unambiguous indicator gene.

Given that the cross-validation procedure produced a range of different basis functions with a range of weights, it is interesting to consider what linear models can

be generated by combining these basis functions and their weights. This should give us some further indication of how important particular basis functions are.

One linear model can be obtained by taking the average weighting of each probe across all of the cross-validation models in which it takes part. The result of applying this to the micro-arrays is presented in Figure 5-3 as the scores produced prior to conversion to probabilities. This model misclassifies only four micro-arrays, giving a 90% accuracy rate.

This model does not take into account that some probes are present in fewer models. It is possible to reflect this by averaging the weights across all models, using a weight of zero where the probe is not used in a particular model. The result of applying this to the micro-arrays is presented in Figure 5-4. This model correctly classifies all of the micro-arrays. However, the associated confidences are lower, as demonstrated by the reduced magnitude of the outputs. The increase in accuracy of this model supports the idea that basis functions which are frequently present in different models are more informative to the classification task.

Using the contribution of just FOS to the model in Figure 5-4 (FOS level multiplied by its weight), all of the samples taken after treatment can be correctly identified, but 11 out of the 20 samples taken before treatment are misclassified. Similarly, using just the contribution of TDO2, all of samples taken after treatment can be correctly identified, but 4 of the samples taken before treatment are incorrectly predicted as being after treatment. This contrasts strongly with the behaviour of the contribution of the third component RGS1, which uniformly predicts all samples as belong to the before treatment class, with just one before and one after treatment sample predicted as after treatment. Each of the models generated during the cross validation procedure

contains exactly one probe that uniformly predicts all microarrays as belonging to one class (data not shown). We propose that the RVM is using these uniform predictors as a calibrated model of the level and variation inherent within this data set.

The aim of this RVM approach is to classify microarrays into two classes using the expression levels associated with each gene within each microarray. This methodology produces models that can be readily interpreted in terms of the contribution of each gene. However, it is not the primary aim of this method to indicate discriminating genes. A student t-test is more appropriate as a means for identifying genes with differential expression levels. This test calculates the probability that two sets of numbers have normal distributions that are distinguishable from one-another.

The student t-test scores associated with the range of levels in the before and after treatment groups is presented in Table 3-2. The column labelled TP contains the t-test scores for the two sets of microarrays taking into account the pairing between samples taken from the same patient before and after treatment. This information was not available to the RVM, and the student t-test scores assuming no such pairing are contained within the column labelled TS.

The probes for FOS and JUN have values that are extremely significant, indicating that there is very strong support for the hypothesis that the microarray levels before and after treatment come from different distributions. Generally, the t-test scores (both TS and TP) do not show any clear trend related to the rank of the probe in the table, or with the use of the probe as a uniform predictor. Although many of the probes used in the cross-validation models do have significant t-test scores, some do not, both at the 5 % and the 1 % significance level. Interestingly, many of the TP

scores are actually worse than the associated TS scores. It would be expected that in a system with low noise, the extra information provided by the sample pairing would lead to systematically greater significance. The presence of counter-examples may indicate that when considering individual genes, the level of noise in this data in some cases obscures the signal provided by the before and after treatment pairing.

Table 5-2 Genes used by cross-validation models

All information taken from the data files providing the expression data. Accession values of (*) indicate that the spot had no associated probe. TS is the value of the student t-test assuming the before and after samples to be unpaired. TP is the value of the student t-test taking into account that before and after samples are paired.

Probe	Accession	Symbol	Uses	TS (%)	TP (%)	Description
9016	AA040944	FOS	40	0.00	0.00	v-fos FBJ murine osteosarcoma viral oncogene homolog
8530	AA027832	HBA2	22	4.89	2.95	Hemoglobin, alpha 2
243	AA017544	RGS1	17	1.11	1.38	Regulator of G-protein signalling 1
2114	AA454668	PTGS1	11	0.24	0.07	Prostaglandin-endoperoxide synthase 1 (prostaglandin G/H synthase and cyclooxygenase)
6333	N50845		11	5.74	9.64	
5635	*		8	0.60	0.39	
6077	AA425316	LOC51700	7	1.78	2.02	Cytochrome b5 reductase b5R.2
7399	AA026682	TOP2A	5	0.71	0.60	Topoisomerase (DNA) II alpha (170kD)
3903	*		3	5.13	4.39	
3901	*		2	0.36	0.45	
5284	T72398	TDO2	2	7.33	0.83	Tryptophan 2,3-dioxygenase
6223	*		2	15.48	16.19	
6494	W96134	JUN	2	0.00	0.00	v-jun avian sarcoma virus 17 oncogene homolog
7956	T63045	IGL@	2	16.01	1.40	Immunoglobulin lambda locus
244	AA074224	RCV1	1	9.98	12.11	Recoverin
2753	*		1	4.63	6.26	
4468	AA453345	JAK2	1	6.99	3.93	Janus kinase 2 (a protein tyrosine kinase)
5002	AA620359		1	1.20	0.62	
6043	H87471	KYNU	1	20.95	4.81	Kynureninase (L-kynurenine hydrolase)
7704	N71028		1	0.99	0.17	
8494	*		1	4.58	8.33	
8719	AA504348	TOP2A	1	2.34	2.39	Topoisomerase (DNA) II alpha (170kD)

Error! Not a valid link.

Figure 5-1 Scatter Plot of the Two Topoisomerase II Probes Used.

There is one point for each of the 40 micro-arrays. The x values are the levels of the probe for AA504348, and the y values are the levels of the probe AA026682. Both of these are probes for the TOP2A gene. The R^2 value is the correlation between the levels measured for these two probes under identical conditions.

Error! Not a valid link.

Figure 5-2 Expression Levels for Each Probe Used

For each probe, there are three bars. Each data-point displays the mean level for a probe across a range of micro-arrays. The error bars display two standard deviations around the mean. In each case, the left-most bar corresponds to the mean and standard deviation of the probes level across the 20 micro-arrays taken before treatment, and the right-most par corresponds to the mean and standard deviations for the probe across the 20 micro-array measurements after drug treatment. Each data-point is labeled with the gene name if present. If this was not present, the accession number is used. If this was not available, the probe number is used. The probes are in the same order as Table 5-2.

Error! Not a valid link.

Figure 5-3 Average Weighs Across Relevant Models.

The samples after treatment are to the left, and samples before treatment are to the right. All prediction values are in the units of the GLM before conversion into probabilities. Values below 0 will map to probabilities below 0.5, and values above 0 will map to probabilities above 0.5. All of the before samples have been correctly classified. Four of the after samples are misclassified, and are indicated with an asterisk (*).

*
* * *

After Before

Error! Not a valid link.

Figure 5-4 Average Weights Across All Models

The samples after treatment are to the left, and samples before treatment are to the right. All prediction values are in the units of the GLM before conversion into probabilities. Values below 0 will map to probabilities below 0.5, and values above 0 will map to probabilities above 0.5. All of the samples have been correctly classified.

After Before

5.4.3 Function of Genes Identified by GLM Models

If the model learned a biologically significant signal, this should be reflected in the probes used to construct the model (as listed in Table 5-2). For many of the genes, this is indeed the case. Several genes known to be involved in the action of doxorubicin are present.

TOP2A directly interacts with doxorubicin, leading to the single-strand break mechanism of drug activity. This appears to be down-regulated in the group after treatment. This could be evidence that the cancers are developing doxorubicin resistance by repressing the TOP2A gene. Alternatively, potentially irreversible interactions between TOP2A and doxorubicin intercalated with DNA, or the relative lack of super-coiling due to many single-strand breaks may be fooling the regulatory mechanisms for TOP2A into behaving as if there are sufficient levels of the protein, leading to down-regulation of the gene.

JUN and FOS are part of the pathway that mediates apoptosis in response to excessive rates of single-strand breakages. Both of these appear to be up-regulated in the group after treatment. JUN and FOS form a transcription regulatory complex, and in cells responding to single-strand break stress, this complex interacts with the genes responsible for activating the apoptosis response. The resulting reduction in level of free JUN and FOS may cause their synthesis to be up-regulated to compensate.

RGS1 is a repressor of the G protein signalling that is involved in the regulation of b-cell activation and proliferation, as well being indicated in a range of cancers[35]. It appears to be marginally down-regulated after treatment. G proteins are involved in a

[35] See http://caroll.vjf.cnrs.fr/cancergene/CG516.html for a description of RGS1

wide range of signalling activities, and initiate MAP-kinase cascades. By down-regulating the repressor, the activity of the G proteins would be enhanced, increasing the strength of the signalling pathway. The single strand breaks introduced by Doxorubicin activity tend to arrest cell division. An increase in proliferation signals mediated by G protein signalling may compensate for this effect.

Two enzymes, TDO2 and KYNU, are present from the tryptophan metabolism pathway. Both of these enzymes appear to be down regulated in response to doxorubicin treatment. TDO2 catalysis the conversion of tryptophan to N-formyl-kynurenine. KYNU catalyses the conversion of this compound to formyl-anthranilate. It is intriguing that the models identified these two enzymes, given their proximity in a pathway. Intracellular levels of tryptophan around tumours have been shown to be abnormal (Iwagaki, Hizuta et al. 1995; Huang, Fuchs et al. 2002), but there is no clear indication of why this pathway should be important in response to doxorubicin.

The two genes HBA2 (haemoglobin alpha 2 subunit) and LOC51700 (cytochrome-b5 reductase) are present in several models. Both of these appear to be down regulated in response to doxorubicin treatment. Cytochrome-b5 and haemoglobin both require haem[36] for their production. Cytochrome-b5 reductase decreases the levels of available cytochrome-b5. Reduction in the levels of this enzyme would lead to increased levels of cytochrome-b5. A down regulation of HBA2 production would reduce the amount of haem becoming incorporated into haemoglobin. If both proteins are expressed within the same cells, these two processes would act together to increase the level of cytochrome-b5.

---

[36] See http://www.genome.ad.jp/dbget-bin/www_bget?compound+C00032 and links from that page for a more full description of haem and the Prophyrin metabolism pathway

As the samples used for microarray analysis were obtained from biopsies, it is inevitable that they represent expression levels from a range of different cell types. It is possible that within this population there were immature red blood cells. Although the nucleated red blood cell precursors are present only in the bone-marrow, there is a stage in their differentiation intermediate between this and mature red blood cells that contains mitochondria and messenger RNA. For a couple of days, these are present in the blood stream (Gilbert 2003). During this maturation stage, haemoglobin is synthesized. It is possible that the chemotherapy results in a decrease rate of red blood cell production. This would lead to a decreased number of maturing red blood cells in the circulatory system, and therefore a lower measured level of the haemoglobin mRNA.

5.5 Conclusions, Applications and Future Work

In this chapter, we have shown that RVMs can be used in the analysis of expression data that contains few samples and is noisy. The RVM was both able to perform the required classification task, and the model produced has clearly identified biologically relevant genes.

Cluster analysis of this expression data does not help in discovering genes that have modified expression levels in response to treatment with doxorubicin. Clustering by correlation co-efficient identifies clusters containing pairs of micro-arrays from a single patient. It does not produce clusters corresponding to all micro-arrays pre- or post-treatment, indicating that the history of the cell line is the primary signal in the expression profiles.

When an SVM was trained using the tumour sample expression data, it appeared to memorize the training set. When the same model was trained using the probabilistic RVM trainer, the RVM rejected the hypothesis that the data was separable using the

linear kernel function. This indicated that with just 20 samples, a conventional SVM could not be constructed to classify these samples into pre- and post-treatment.

Using an alternative strategy, an RVM was constructed with one basis function for each unique probe used to measure the level of a gene, and applied to learn a discriminator to predict whether tumour samples were pre- or post-treatment. It was able to learn signals that correlated with the treatment status. The function learned by RVM did appear to display all of the expected traits of sparsity, simplicity and generalization expected from this training method. Of course, given 8,102 genes to choose from, a model could trivially be constructed that performed very well on the training set. However, this would not be expected to generalize to unseen data. The results of the cross-validated training indicate that the models do generalize regardless of the sub-set used for training and testing, and that the models do not purely contain some statistically aberrant signal present by chance in the training set. With larger training sets, it should be possible to learn models with better estimates for which genes are informative and are less prone to over-fitting.

Some of the probes identified as basis functions during cross-validation appear to show differences in their average levels before and after drug treatment. Others do not. However, the RVMs are not looking at genes in isolation, but rather looking at interactions between them. A number of genes were identified as indicators that make clear biological sense, given the known action of doxorubicin (JUN, FOS, topoisomerase II). A number of others are not surprising, such as those associated with signalling cascades. Others, such as TDO2 and KYNU are implicated by their presence in the model and their differences in mean level before and after treatment.

A final group appear to be used by the model as a measure of the noise in the system, or to obtain a baseline from which all other levels can be calculated.

This use of RVMs is potentially applicable to any situation where large numbers of expression levels have been measured and a test is required which will indicate which of these are informative for a particular biological response. The classifier generated can be used to classify new data. RVMs can be trained using any set of basis functions. This is applicable to a wide range of situations, for example, screening expression levels from patient samples to estimate which of a range of anti-cancer drugs may be an appropriate treatment. It is possible to combine expression data with any other measurements. For example, expression levels could be combined with information about the presence or absence of SNPs, direct biological measurements, such as pulse or breathing rate, and so forth into a single predictive model.

The RVMs have two advantages over support vector machines (SVM) for this type of data. Firstly, to evaluate an SVM it is necessary to calculate products for every gene on the micro-array. The SVM will be invalid if all of the genes on the original micro-array are not also measured in the clinical sample from the patient, as the full dot product between the support vectors and the sample cannot be calculated. In addition, this requires one multiplication per micro-array spot per support vector. An RVM of the form described above only requires that the genes that are used by the model be within the set measured in the patient sample. It requires one multiplication for each gene that is used as a basis function to the learned weight.

Secondly, the GLMs produced by RVMs give an indication of confidence in their prediction. This potentially allows the person interpreting the model output to make sensible judgements about how to use the model's prediction (for example, ignoring

predictions with very low confidence). SVMs give an output value, but the absolute scale of this number is dependant on the distance from the separating hyper-plane of the two support vectors corresponding to the closest correctly classified data points from each class. Two SVMs using different support vectors will have incomparable scales, making it impossible to compare these values directly.

To evaluate the effectiveness of these RVMs for other classification tasks using micro-array data, more data sets need to be analysed. The data set used here was both small and noisy. It is to be expected that with larger data sets containing cleaner expression levels, that much higher levels of classification accuracy can be achieved. In the case where this method is used as a way of identifying biologically relevant genes, methods need to be developed to extract models with more genes. This could be achieved by training the model repeatedly, removing all probes identified by previous models until the model does not perform the classification task. Alternatively, it may be possible to look at the information each indicator gene is contributing, and to use some form of hierarchical or single-linkage clustering to identify those with patterns of expression that share information with it. The RVM could be modified to indicate if each basis was rejected because it did not contribute to the accuracy of the model, or because it duplicated information present in another basis.

Since this work was carried out, related relevance-based approaches have begun to emerge, for example (Li, Campbell et al. 2002) and Gene-Rave[37]. However, these

[37] See http://www.bioinformatics.csiro.au/GeneRave/products.html and the examples link from this page

methods do not seem to be producing results with quite the same high level of sparsity our method generates. Neither of these methods has as yet solved the question of how to retrieve the indicator genes removed from the model because they give information correlated to that of the selected relevant genes.

Concluding Remarks

The vast volumes of biological data being produced now overwhelm the traditional paradigm of individual scientists studying individual results and making and testing individual hypotheses. In this dissertation, I present tools and methods that allow data sets of genomic scales to be explored, analyzed and learned from. The BioJava project provides programming tools for manipulating genomic data sets. HMMs can be constructed which leverage un-supervised learning techniques to elucidate the inherent structure of chromosomes. SVMs and latterly RVMs can be used to perform regression and classification tasks on large quantities data with an unprecedented degree of sparsity and generalization. Here, they are used for the diverse tasks of predicting recombination rates and classifying tumour samples into those treated and un-treated with Doxorubicin.

During my PhD studies, I have used BioJava and its machine learning implementations in a range of other situations, which are not discussed in detail here. This was partly to define the limitations of the methods and partly for scientific exploration. Briefly, SVMs were applied to a wide range of regression and classification tasks. These included the implementation of an e-mail spam filter, assessing the accuracy of gene predictions given the outputs of multiple programs and curve smoothing for recombination rates. RVMs were applied to an equally wide range of problems, including predicting protein secondary structure elements, sequence comparison using HMM kernels and simultaneous estimation of expression profile class and promoter structures. HMMs were used to model 3-D DNA structure using a multinomial Gaussian emission state, Gibbs-sampling of expression profiles, promoter finding, protein secondary structure prediction by pair wise alignment and

HMM-based kernel functions. This is by no means an exhaustive list of mini-projects undertaken within the past four years, but gives a flavour of the range of problems that can be tackled using these technologies.

Since this thesis was written, BioJava has continued to develop (as discussed in chapter 2), demonstrating that the original APIs are both flexible and sufficient, allowing a wide degree of reuse and extension.

None of the methods presented here are limited to the problems to which they were applied. The task ahead is to use these and other technologies to make new discoveries about how genomes are structured, function, evolve and fail. The possible applications are wide-ranging; medicine, agriculture, bio-engineering, palaeontology, to name but a few. I look forward to seeing where this leads us.

Forgive us for what we have done and what we have left undone

(Extract from the Anglican Order of Service)

References

Aho, A. V., R. Sethi, et al. (1985). Compilers: Principles, Techniques, and Tools, Addison-Wesley.

Alizadeh, A. A., M. B. Eisen, et al. (2000). "Distinct types of diffuse large B-cell lymphoma identified by gene expression profiling." Nature **403**(6769): 503-11.

Altschul, S. F., W. Gish, et al. (1990). "Basic local alignment search tool." J Mol Biol **215**(3): 403-10.

Asai, K., S. Hayamizu, et al. (1993). "Prediction of protein secondary structure by the hidden Markov model." Comput Appl Biosci **9**(2): 141-6.

Bailey, T. L. and C. Elkan (1994). "Fitting a mixture model by expectation maximization to discover motifs in biopolymers." Proc Int Conf Intell Syst Mol Biol **2**: 28-36.

Bairoch, A. (2000). "The ENZYME database in 2000." Nucleic Acids Res **28**(1): 304-5.

Bateman, A., E. Birney, et al. (2000). "The Pfam protein families database." Nucleic Acids Res **28**(1): 263-6.

Bates, M. D., C. R. Erwin, et al. (2002). "Novel genes and functional relationships in the adult mouse gastrointestinal tract identified by microarray analysis." Gastroenterology **122**(5): 1467-82.

Baum, L. E., T. Petrie, et al. (1970). "A maximization technique occuring in the statistical analysis of probabilitic functions of Markov chains." The Annals of Mathematical Statistics **41**(1): 164171.

Benson, D. A., I. Karsch-Mizrachi, et al. (2003). "GenBank." Nucleic Acids Res **31**(1): 23-7.

Birney, E. and R. Durbin (1997). "Dynamite: a flexible code generating language for dynamic programming methods used in sequence comparison." Proc Int Conf Intell Syst Mol Biol **5**: 56-64.

Birney, E. and R. Durbin (2000). "Using GeneWise in the Drosophila annotation experiment." Genome Res **10**(4): 547-8.

Bishop, C. M. and T. M. E. (2000). Variational relevance vector machines. Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann.

Boeckmann, B., A. Bairoch, et al. (2003). "The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003." Nucleic Acids Res **31**(1): 365-70.

Bowman, S., D. Lawson, et al. (1999). "The complete nucleotide sequence of chromosome 3 of Plasmodium falciparum." Nature **400**(6744): 532-8.

Brenton, J. D., S. A. Aparicio, et al. (2001). "Molecular profiling of breast cancer: portraits but not physiognomy." Breast Cancer Res **3**(2): 77-80.

Brown, M., R. Hughey, et al. (1993). "Using Dirichlet mixture priors to derive hidden Markov models for protein families." Proc Int Conf Intell Syst Mol Biol **1**: 47-55.

Buck, K. J., R. A. Harris, et al. (1991). "A general method for quantitative PCR analysis of mRNA levels for members of gene families: application to GABAA receptor subunits." Biotechniques **11**(5): 636-41.

Burge, C. and S. Karlin (1997). "Prediction of complete gene structures in human genomic DNA." J Mol Biol **268**(1): 78-94.

Butte, A. J. and I. S. Kohane (2000). "Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements." Pac Symp Biocomput: 418-29.
Butte, A. J., J. Ye, et al. (2001). "Determining significant fold differences in gene expression analysis." Pac Symp Biocomput: 6-17.
Chen, T., H. L. He, et al. (1999). "Modeling gene expression with differential equations." Pac Symp Biocomput: 29-40.
Chu, S., J. DeRisi, et al. (1998). "The transcriptional program of sporulation in budding yeast." Science **282**(5389): 699-705.
Churchill, G. A. (1989). "Stochastic models for heterogeneous DNA sequences." Bull Math Biol **51**(1): 79-94.
Corcoran, L. M., J. K. Thompson, et al. (1988). "Homologous recombination within subtelomeric repeat sequences generates chromosome size polymorphisms in P. falciparum." Cell **53**(5): 807-13.
Dawson, E., G. R. Abecasis, et al. (2002). "A first-generation linkage disequilibrium map of human chromosome 22." Nature **418**(6897): 544-8.
Dib, C., S. Faure, et al. (1996). "A comprehensive genetic map of the human genome based on 5,264 microsatellites." Nature **380**(6570): 152-4.
Dixon, D. A. and S. C. Kowalczykowski (1991). "Homologous pairing in vitro stimulated by the recombination hotspot, Chi." Cell **66**(2): 361-71.
Dowell, R. D., R. M. Jokerst, et al. (2001). "The Distributed Annotation System." BMC Bioinformatics **2**(1): 7.
Down, T. (2003). Genetics. Cambridge, Cambridge.
Dunham, I., N. Shimizu, et al. (1999). "The DNA sequence of human chromosome 22." Nature **402**(6761): 489-95.
Durbin, R., Eddy, E., Krogh A. and Mitchison, G. (1998). Biological Seqence Analysis. Cambridge, UK, Cambridge University Press.
Eddy, S. R. (2001). "Profile hidden markov models for biological sequence analysis." Bioinformatics **14**: 755-763.
Eisen, M. B., P. T. Spellman, et al. (1998). "Cluster analysis and display of genome-wide expression patterns." Proc Natl Acad Sci U S A **95**(25): 14863-8.
Escalante, A. A., A. A. Lal, et al. (1998). "Genetic polymorphism and natural selection in the malaria parasite Plasmodium falciparum." Genetics **149**(1): 189-202.
Falquet, L., M. Pagni, et al. (2002). "The PROSITE database, its status in 2002." Nucleic Acids Res **30**(1): 235-8.
Figueiredo, L. M., L. H. Freitas-Junior, et al. (2002). "A central role for Plasmodium falciparum subtelomeric regions in spatial positioning and telomere length regulation." Embo J **21**(4): 815-24.
Gamma, E., R. Helm, et al. (1994). Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional.
Gardner, M. J., N. Hall, et al. (2002). "Genome sequence of the human malaria parasite Plasmodium falciparum." Nature **419**(6906): 498-511.
Gardner, M. J., H. Tettelin, et al. (1999). "The malaria genome sequencing project: complete sequence of Plasmodium falciparum chromosome 2." Parassitologia **41**(1-3): 69-75.
Gendrel, C. G., A. Boulet, et al. (2000). "(CA/GT)(n) microsatellites affect homologous recombination during yeast meiosis." Genes Dev **14**(10): 1261-8.
Gilbert, S. F. (2003). Developmental Biology. Sunderland, MA, Sinauer Associates, Inc.

Gosling, J., B. Joy, et al. (2000). The Java Language Specification, Addison-Wesley.
Grandjean, F., L. Bremaud, et al. (2001). "Sequential gene expression of P-glycoprotein (P-gp), multidrug resistance-associated protein (MRP) and lung resistance protein: functional activity of P-gp and MRP present in the doxorubicin-resistant human K562 cell lines." Anticancer Drugs **12**(3): 247-58.
Grundy, W. N., T. L. Bailey, et al. (1997). "Meta-MEME: motif-based hidden Markov models of protein families." Comput Appl Biosci **13**(4): 397-406.
Guillouzic, S., I. I. L'Heureux, et al. (2000). "Rate processes in a delayed, stochastically driven, and overdamped system." Phys Rev E Stat Phys Plasmas Fluids Relat Interdiscip Topics **61**(5A): 4906-14.
Guo, Z., R. A. Guilfoyle, et al. (1994). "Direct fluorescence analysis of genetic polymorphisms by hybridization with oligonucleotide arrays on glass supports." Nucleic Acids Res **22**(24): 5456-65.
Hall, N., A. Pain, et al. (2002). "Sequence of Plasmodium falciparum chromosomes 1, 3-9 and 13." Nature **419**(6906): 527-31.
Hasan, S. (2003). The Sanger Centre. Cambridge.
Huang, A., D. Fuchs, et al. (2002). "Serum tryptophan decrease correlates with immune activation and impaired quality of life in colorectal cancer." Br J Cancer **86**(11): 1691-6.
Hubbard, T., D. Barker, et al. (2002). "The Ensembl genome database project." Nucleic Acids Res **30**(1): 38-41.
Hughes, T. R., M. Mao, et al. (2001). "Expression profiling using microarrays fabricated by an ink-jet oligonucleotide synthesizer." Nat Biotechnol **19**(4): 342-7.
Hughey, R. and A. Krogh (1995). SAM: Sequence alignment and modeling software system. Santa Cruz, CA, University of California.
Iwagaki, H., A. Hizuta, et al. (1995). "Decreased serum tryptophan in patients with cancer cachexia correlates with increased serum neopterin." Immunol Invest **24**(3): 467-78.
Kihara, C., T. Tsunoda, et al. (2001). "Prediction of sensitivity of esophageal tumors to adjuvant chemotherapy by cDNA microarray analysis of gene-expression profiles." Cancer Res **61**(17): 6474-9.
Krogh, A., M. Brown, et al. (1994). "Hidden Markov models in computational biology. Applications to protein modeling." J Mol Biol **235**(5): 1501-31.
Lashkari, D. A., J. L. DeRisi, et al. (1997). "Yeast microarrays for genome wide parallel genetic and gene expression analysis." Proc Natl Acad Sci U S A **94**(24): 13057-62.
Lawrence, T. S. (1988). "Reduction of doxorubicin cytotoxicity by ouabain: correlation with topoisomerase-induced DNA strand breakage in human and hamster cells." Cancer Res **48**(3): 725-30.
Lawrence, T. S. and M. A. Davis (1990). "The influence of Na+,K(+)-pump blockade on doxorubicin-mediated cytotoxicity and DNA strand breakage in human tumor cells." Cancer Chemother Pharmacol **26**(3): 163-7.
Li, Y., C. Campbell, et al. (2002). "Bayesian automatic relevance determination algorithms for classifying gene expression data." Bioinformatics **18**(10): 1332-9.
Liefers, G. J. and R. A. Tollenaar (2002). "Cancer genetics and their application to individualised medicine." Eur J Cancer **38**(7): 872-9.

Lindholm, T. and F. Yellin (1999). The Java Virtual Machine Specification, Addison-Wesley.

Majewski, J. and J. Ott (2000). "GT repeats are associated with recombination on human chromosome 22." Genome Res **10**(8): 1108-14.

Mody, M., Y. Cao, et al. (2001). "Genome-wide gene expression profiles of the developing mouse hippocampus." Proc Natl Acad Sci U S A **98**(15): 8862-7.

Nedelman, J., P. Heagerty, et al. (1992). "Quantitative PCR with internal controls." Comput Appl Biosci **8**(1): 65-70.

Nelder, J. A. and P. McCulagh (1983). Generalized Linear Models. London, Chapman and Hall.

Ning, Z., A. J. Cox, et al. (2001). "SSAHA: a fast search method for large DNA databases." Genome Res **11**(10): 1725-9.

O'Hagan, A. (1994). Bayesian Inference. London, Hodder Anold.

Perou, C. M., T. Sorlie, et al. (2000). "Molecular portraits of human breast tumours." Nature **406**(6797): 747-52.

Platt, J. (1998). Fast Training of Support Vector Machines using Sequential Minimal Optimization. Advances in Kernel Methods - Support Vector Learning. B. C. a. S. Scholkopf B., A., MIT Press.

Pocock, M. R., T. Down, et al. (2000). "BioJava: Open Source Components for Bioinformatics." sigbio newsletter **20**(2): 10-12.

Pollack, Y., A. L. Katzen, et al. (1982). "The genome of Plasmodium falciparum. I: DNA base composition." Nucleic Acids Res **10**(2): 539-46.

Potmesil, M., Y. H. Hsiang, et al. (1988). "Resistance of human leukemic and normal lymphocytes to drug-induced DNA cleavage and low levels of DNA topoisomerase II." Cancer Res **48**(12): 3537-43.

Pourquier, P., D. Montaudon, et al. (1998). "Doxorubicin-induced alterations of c-myc and c-jun gene expression in rat glioblastoma cells: role of c-jun in drug resistance and cell death." Biochem Pharmacol **55**(12): 1963-71.

Rabiner, L. R. (1989). "A tutorial on Hidden Markov Models and selected applications in speech recognition." Proceedings of the IEEE **77**(2): 257-286.

Ramaswamy, S., P. Tamayo, et al. (2001). "Multiclass cancer diagnosis using tumor gene expression signatures." Proc Natl Acad Sci U S A **98**(26): 15149-54.

Rampone, S. (1998). "Recognition of splice junctions on DNA sequences by BRAIN learning algorithm." Bioinformatics **14**(8): 676-84.

Reinhardt, A. and T. Hubbard (1998). "Using neural networks for prediction of the subcellular location of proteins." Nucleic Acids Res **26**(9): 2230-6.

Rice, P., I. Longden, et al. (2000). "EMBOSS: the European Molecular Biology Open Software Suite." Trends Genet **16**(6): 276-7.

Rooney, D. E. (2001). Human Cytogenetics: Constitutional Analysis, A Practical Approach. Oxford, Oxford University Press.

Rost, B. and C. Sander (1994). "Combining evolutionary information and neural networks to predict protein secondary structure." Proteins **19**(1): 55-72.

Schena, M., D. Shalon, et al. (1995). "Quantitative monitoring of gene expression patterns with a complementary DNA microarray." Science **270**(5235): 467-70.

Shalon, D. (1998). "Gene expression micro-arrays: a new tool for genomic research." Pathol Biol (Paris) **46**(2): 107-9.

Sjolander, K., K. Karplus, et al. (1996). "Dirichlet mixtures: a method for improved detection of weak but significant protein sequence homology." Comput Appl Biosci **12**(4): 327-45.

Smith, T. F. and M. S. Waterman (1981). "Identification of common molecular subsequences." J Mol Biol **147**(1): 195-7.

Smolen, P., D. A. Baxter, et al. (2001). "Modeling circadian oscillations with interlocking positive and negative feedback loops." J Neurosci **21**(17): 6644-56.

Stajich, J. E., D. Block, et al. (2002). "The Bioperl toolkit: Perl modules for the life sciences." Genome Res **12**(10): 1611-8.

Stoesser, G., W. Baker, et al. (2003). "The EMBL Nucleotide Sequence Database: major new developments." Nucleic Acids Res **31**(1): 17-22.

Tewey, K. M., T. C. Rowe, et al. (1984). "Adriamycin-induced DNA damage mediated by mammalian DNA topoisomerase II." Science **226**(4673): 466-8.

Tipping, M. E. (2000). The Relevance Vector Machine. Advances in Neural Information Processing Systems 12, MIT Press.

van 't Veer, L. J., H. Dai, et al. (2002). "Gene expression profiling predicts clinical outcome of breast cancer." Nature **415**(6871): 530-6.

Vapnik, V. N. (1995). The Nature of Statistical Learning Theory, Springer.

Womble, D. D. (2000). "GCG: The Wisconsin Package of sequence analysis programs." Methods Mol Biol **132**: 3-22.