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

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

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

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

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

$$k \in I$$
$$e_k(a) = p(a \in \Omega \mid k)$$
$$t_k(j) = p(j \mid k) \forall (k, j) \in \Sigma$$

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.