Title
Analysis and applications of conserved sequence patterns in proteins

Permalink
https://escholarship.org/uc/item/5kh9z3fr

Author
Ie, Tze Way Eugene

Publication Date
2007

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA, SAN DIEGO

Analysis and Applications of Conserved Sequence Patterns in Proteins

A dissertation submitted in partial satisfaction of the requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Tze Way Eugene Ie

Committee in charge:

Professor Yoav Freund, Chair
Professor Sanjoy Dasgupta
Professor Charles Elkan
Professor Terry Gaasterland
Professor Philip Papadopoulos
Professor Pavel Pevzner

2007
The dissertation of Tze Way Eugene Ie is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2007
DEDICATION

This dissertation is dedicated in memory of my beloved father,
Ie It Sim (1951–1997).
# Table of Contents

Signature Page ........................................ iii
Dedication ........................................ iv
Table of Contents ..................................... v
List of Figures ....................................... viii
List of Tables ......................................... x
Acknowledgements ................................... xi
Vita, Publications, and Fields of Study ........... xiii
Abstract of the Dissertation ......................... xv

1 Introduction ........................................ 1
  1.1 Protein Homology Search ............................ 1
  1.2 Sequence Comparison Methods .................... 2
  1.3 Statistical Analysis of Protein Motifs ............ 4
  1.4 Motif Finding using Random Projections .......... 5
  1.5 Microbial Gene Finding without Assembly ....... 7

2 Multi-Class Protein Classification using Adaptive Codes ................. 9
  2.1 Profile-based Protein Classifiers ................ 14
  2.2 Embedding Base Classifiers in Code Space ....... 15
  2.3 Learning Code Weights ............................ 18
    2.3.1 Ranking Perceptrons ......................... 18
    2.3.2 Structured SVMs .............................. 20
  2.4 Data Set for Training Base Classifiers .......... 21
    2.4.1 Superfamily Detectors ....................... 21
    2.4.2 Fold Detectors ............................... 22
    2.4.3 Data Set for Learning Weights .............. 22
  2.5 Experimental Setup ................................ 23
    2.5.1 Baseline Methods ............................ 23
    2.5.2 Extended Codes .............................. 24
    2.5.3 Previous Benchmark .......................... 24
  2.6 Code Weight Learning Algorithm Setup .......... 25
    2.6.1 Ranking Perceptron on One-vs-all Classifier Codes 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.2</td>
<td>Structured SVM on One-vs-all Classifier Codes</td>
<td>25</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Ranking Perceptron on One-vs-all Classifier Codes with PSI-BLAST Extension</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>Results</td>
<td>26</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Fold and Superfamily Recognition Results</td>
<td>26</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Importance of Hierarchical Labels</td>
<td>26</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Comparison with Previous Benchmark</td>
<td>30</td>
</tr>
<tr>
<td>2.8</td>
<td>Discussion</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>Statistical Analysis of Protein Motifs: Application to Homology Search</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Related Work</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Statistical Significance of Conserved Patterns</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>Pattern Refinement</td>
<td>40</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Estimating PSSMs Using Mixture Models</td>
<td>42</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Estimating PSSMs Using Statistical Tests</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Defining Motif Families</td>
<td>49</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Motifs and Zones</td>
<td>49</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Similarity Matrices for Sequence Motifs</td>
<td>50</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Clustering Motifs</td>
<td>50</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Motif Templates</td>
<td>51</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Zone Templates</td>
<td>53</td>
</tr>
<tr>
<td>3.5</td>
<td>Sequence Search Engine</td>
<td>55</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Building BioSpike</td>
<td>55</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Query Processing</td>
<td>55</td>
</tr>
<tr>
<td>3.6</td>
<td>Results</td>
<td>57</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Comparison with PSI-BLAST</td>
<td>57</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Results from specific searches</td>
<td>59</td>
</tr>
<tr>
<td>3.7</td>
<td>Conclusion</td>
<td>61</td>
</tr>
<tr>
<td>4</td>
<td>Motif Finding using Random Projections</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>Background and Related Work</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Our Contributions</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>Sequence Embedding</td>
<td>67</td>
</tr>
<tr>
<td>4.4</td>
<td>Random Linear Projections</td>
<td>69</td>
</tr>
<tr>
<td>4.5</td>
<td>Motif Indexing Framework</td>
<td>71</td>
</tr>
<tr>
<td>5</td>
<td>Gene Finding in Microbial Genome Fragments</td>
<td>74</td>
</tr>
<tr>
<td>5.1</td>
<td>Gene Finding for Metagenomics</td>
<td>76</td>
</tr>
<tr>
<td>5.2</td>
<td>AdaBoost</td>
<td>77</td>
</tr>
<tr>
<td>5.3</td>
<td>Alternating Decision Trees</td>
<td>78</td>
</tr>
<tr>
<td>5.4</td>
<td>Problem Setup</td>
<td>80</td>
</tr>
</tbody>
</table>
Figure 2.1  The main stages of the code weights learning methodology for protein classification. ........................................... 12

Figure 2.2  The mechanism for learning code weights. ............... 13

Figure 2.3  The embedding of a protein sequence discriminant vector in unweighted code space (top). The embedding of the same protein sequence discriminant vector in a weighted code space (bottom).  . 17

Figure 2.4  Pseudocode for the ranking perceptron algorithm used to learn code weighting. ........................................... 19

Figure 2.5  A hypothetical SCOP structure for illustrating data set construction, showing SCOP folds, superfamilies, and families.  . 22

Figure 3.1  The tails of the estimated null and empirical score inverse cumulative distributions for the PSSM corresponding to the RuvA BLOCK motif in Bacterial DNA recombination protein (InterPro BLOCK Id:IPB000085A). ........................................... 41

Figure 3.2  Algorithm for computing the estimated empirical model for a PSSM column and the list of amino acid with $\delta$-distributions in the model. ........................................... 44

Figure 3.3  Relationships between the amino acids. ................. 46

Figure 3.4  Greedy algorithm for constructing template $T$. ...... 52

Figure 3.5  A comparison of significance of detections and UniProtKB protein coverage between BLOCKS derived profiles and refined template profiles. ........................................... 54

Figure 3.6  The intersection size between PSI-BLAST returned sequences ($P$) and BioSpike returned sequences ($B$). ............... 58

Figure 3.7  The multiple alignment of pattern regions for PDB annotated sequences returned by BioSpike. The fragment labeled $q$ is from the query sequence. ........................................... 59

Figure 3.8  Legume lectin domains in several BioSpike result sequences. The protein sequences are colored in blue, and the detected Legume lectin domains are highlighted in red. Related PDB chains are colored in green. . . . 60
Figure 4.1 **Two randomly chosen projections of the same high dimensional data points.** The blue boxes indicate that the distribution of data points on the one dimensional line is dependent upon the projection.

Figure 4.2 **Random projection algorithm for genomic sequences.** In each successive stage (top to bottom), we fix random positions in the 8-mer patterns. We show that the buckets become more specific in each iteration.

Figure 4.3 **An embedding of amino acids in 3-space.** The hydrophobic residues (mostly on the left) are clearly separated from the hydrophilic residues (mostly on the right). The symbol X is standard for denoting unknown residues in sequence data. It is clearly positioned in the center for all residues.

Figure 4.4 **Histogram comparisons between the empirical and null models.** The empirical model is based on random projection values for length 30 fragments in a database, where 20 of the 30 positions are fixed for sampling and each residue is mapped to a point in $\mathbb{R}^5$.

Figure 4.5 **KL analysis of fixed and free positions along a motif model.** Vertical red dashed lines signify the fixed positions (top) and the motif model is summarized in a frequency matrix (bottom).

Figure 5.1 **The AdaBoost algorithm.**

Figure 5.2 **An ADT for the problem of classifying in-frame codons from out-of-frame codons after 10 iterations of boosting.**

Figure 5.3 **Classification problem for metagenomics gene finding.** The position with the first 0 is an annotated gene start. Circled positions are positive examples and rhombus-marked positions are negative examples.

Figure 5.4 **Describing features on metagenomics fragments.** $(x_i, y_i, w_i)$ describes example $x_i$ with label $y_i$ and initialized with weight $w_i$. The weak learner, $h$, scores the example based on a particular feature. It will be combined with other weak learners in boosting to provide a final classification.

Figure 5.5 **Hypothetical distribution of gene coding patterns in E. coli.** The diagram on the left illustrates a generalized codon distribution over all the proteins. The diagram on the right illustrates groups of proteins found after clustering codon usage patterns.

Figure 5.6 **Comparison with MetaGene.**
Table 2.1 Multi-Class fold and superfamily recognition error rates using full-length and PSI-BLAST extended codes via the ranking perceptron and an instance of SVM-Struct, compared to baseline methods. ......................................................... 27

Table 2.2 Multi-Class fold recognition error rates. This experiment uses fold-only codes via the ranking perceptron and an instance of SVM-Struct, compared to baseline methods. ......................................................... 27

Table 2.3 Multi-Class superfamily recognition error rates. This experiment uses superfamily-only codes via the ranking perceptron and an instance of SVM-Struct, compared to baseline methods. ......................................................... 28

Table 2.4 A comparison of multi-class fold classification average accuracy using PSI-BLAST nearest neighbor methods and using ranking perceptron codes learning algorithm with full-length codes. ............... 29

Table 2.5 Multi-Class fold recognition using codes that incorporate different taxonomic information. The ranking perceptron algorithm is configured to optimize for balanced loss. ......................................................... 30

Table 2.6 Application of code weight learning algorithm on benchmark data set. SVM-2 is the previous benchmark’s best classifier. The ranking perceptron is trained using balanced loss. ......................................................... 31

Table 3.1 BioSpike keywords-directed search statistics. Patterns are ranked by lowest coverage first, and k is the number of pattern “keywords” to incorporate in a protein database search. ......................................................... 58

Table 5.1 Experiment with simulated metagenomics data set. ............ 92
I would like to thank everyone who supported me intellectually and emotionally during my years of graduate school at Columbia University and University of California, San Diego. I am indebted to all of them.

First of all, I would like to thank my advisor and friend, Prof. Yoav Freund, for his tireless support and encouragement throughout my Ph.D. years. He is a constant source of inspiration and this dissertation would not be possible without his guidance. I would also like to thank Prof. Christina Leslie for advising me ever since my undergraduate years at Columbia. I am very lucky to have worked with both of them.

My warm thanks also goes to Dr. Jason Weston and Prof. William Stafford Noble. The work presented in Chapter 2 was done in collaboration with them. It was also my great honor to be mentored by Prof. Jonathan Gross, Prof. Angelos Keromytis, and Prof. Henryk Woźniakowski during my early years of graduate school at Columbia. I would also like to acknowledge all my fellow graduate students at the Freund Lab and the UCSD CSE department. I am indebted to my friends and colleagues at Columbia and UCSD, Aaron Arvey, William Beaver, Rui Kuang, Prof. Alex Biliris, Prof. John Ioannidis; and my friends back home Areta Chan, Chris Cheng, Deborah Leung and Winona Ou to name a few.

Finally, I would like to thank my family and other great friends for their unconditional support. My brother and sister, Andrew Ie and Amanda Ie, kept the spirit in me throughout graduate studies and my mother, Khim Ie, kept me sane through the times while I was away from her. I am greatly indebted to Stanley Pong for his unending support and encouragement. They have all helped me through many difficult times and shared joy during my many years of graduate school. Without them, I could not have completed this dissertation.

Chapter 2, in part, has been a reprint of the material as it appears in E. Ie and J. Weston and C. Leslie and W. S. Noble, “Multi-Class Protein Fold Recognition using Adaptive Codes”, in Proceedings of the 22nd International Conference on
Machine Learning, 2005. The dissertation author was the primary author, and the co-authors listed in this publication supervised the research.

Chapter 3, in part, has been a reprint of the material as it appears in E. Ie and Y. Freund, “BioSpike: Efficient search for homologous proteins by indexing patterns”, in University of California, San Diego, CSE Technical Reports CS2006-0858, 2006. The dissertation author was the primary author, and the co-author listed in this publication supervised the research.
VITA

1981 Born, Hong Kong SAR

Fall 1998 Moved to New York, New York

2001–2004 Teaching Assistant
Columbia University

2002 B.S. in Computer Science
Columbia University

2003 M.S. in Computer Science
Columbia University

2003–2005 Research Assistant
Columbia University

2004–2005 Ph.D. studies in Computer Science
Columbia University

Summer 2005 Engineering Intern
Google, Inc.

Fall 2005 Transferred to University of California, San Diego

2007 Ph.D. in Computer Science
University of California, San Diego

PUBLICATIONS


Modern sequencing initiatives have uncovered a large number of protein sequence data. The exponential growth of these databases are not matched by the rate at which we are annotating them. Reliable structural and functional annotations for protein sequences are limited, and computational methods have been steadily developed to bridge this knowledge gap. This dissertation develops a number of computational techniques for analyzing protein and genomic sequences. They are based heavily on the use of statistics and modern machine learning algorithms.

First, we introduce an application of support vector machines and structured output codes for the problem of discriminating protein sequences into one of many protein structural groups. Although our method works with any type of base binary classifiers, we found that it works best when the base classifiers leverage unlabeled protein sequences. The need to quickly identify similar protein sequences motivates our next contribution, an indexed-based approach to protein search.

We develop a novel indexed-based framework to protein sequence search. The search index is based on robust statistical models of conserved sequence patterns. The user of our system can essentially plug in any existing protein motif libraries to increase the coverage of the index. Furthermore, the framework can systematically refine any bootstrapped profile patterns using large amounts of unannotated sequence data available today. We further supplement the system with a novel
random projections-based algorithm for finding motifs that are prevalent across many protein sequences.

Finally, we outline a new computational problem of finding protein coding regions in microbial genome fragments. This is of particular interest to recent explorations in metagenomics where the microbial communities under scrutiny are increasingly complex. Highly complex metagenomes usually observe lower sequence redundancy for the same amount of sequencing, rendering fragment assembly as an infeasible pre-processing step. We develop a novel evidence integration approach for finding genes on metagenomics fragments requiring no fragment assembly.
1

Introduction

1.1 Protein Homology Search

At the time of writing, there are about 3 million protein sequences in the UniProtKB database [7]. Of these, 7% have comprehensive annotations in the Swiss-Prot knowledgebase [12] and only 1% have a solved 3D structure in the Protein Data Bank (PDB) [10]. This large gap in our biological knowledge and is expected to grow as we sequence more genomes.

To bridge this void, biologists are increasingly using “protein search engines.” The input to such a search engine is a “query” in the form of a protein sequence. The output is a list of protein sequences which are similar to the query sequence in a biologically meaningful way. Many statistical, homology-based methods have been developed for this problem. The three major categories include: pairwise sequence comparison algorithms [3, 82], generative models for protein families [52, 70], and discriminative classifiers [47, 57, 59].

In the context of binary remote homology detection (prediction of whether a protein belongs to a single structural class or not), it has been shown that discriminative classifiers, such as support vector machines (SVMs), outperform the other two types of approaches [56]. In Chapter 2, we introduce a novel approach to combine these predictive binary classifiers for handling the multi-class problem of classifying a protein into one of many structural classes. Our method learns relative weights between one-vs-all classifiers and encodes information about protein
structural hierarchy for multi-class prediction.

The strong performance shown by the discriminative binary protein classifiers is largely attributable to the use of unlabeled protein data [91, 53]. When training and using these classifiers, each input protein sequence must be processed by a basic sequence comparison program, such as PSI-BLAST [4], to find homologous proteins from a large protein database. This pre-processing search is critical as the homologous proteins are subsequently used to provide a richer input representation of the original protein sequence. However, currently available sequence comparison programs are often too slow to be used in a large scale manner.

1.2 Sequence Comparison Methods

In principle, the amino acid sequence determines all the properties of a protein, and these properties can be calculated *ab initio*, that is, using first principles from physics and chemistry. However, *ab initio* methods have had very limited success in practice. An approach that has had more success is to determine protein properties by comparing amino acid sequences using sequence alignments. The assumption, which holds true in most cases, is that proteins with similar amino acid sequences are probably related in structure and/or function.

While high sequence similarity implies structural and functional similarity, the reverse is not true. Proteins with identical structure often consist of very different sequences. This is because neutral mutations are common and residues with similar physicochemical properties often replace one another. Nevertheless, experience from multiple alignments shows us that there are usually regions of 30–80 consecutive residues within proteins that remain highly conserved even when the other parts of the sequence diverge. These conservation patterns are called *sequence motifs*, and are considered the “building blocks” of proteins.

The Smith-Waterman score measures the similarity between two sequences by locally aligning subsequences and computing a value that is a function of the matched and unmatched pairs. The Smith-Waterman algorithm has quadratic
time complexity, $O(mn)$, where $m$ is the number of residues in a protein database and $n$ is the number of residues in the query. With $m$ growing into the billions for modern databases, this algorithmic complexity make any searches intractable.

With the introduction of BLAST, Smith-Waterman-based sequence comparison over large sequence databases became more feasible. BLAST employs smart heuristics that restrict the application of the expensive Smith-Waterman algorithm to a smaller set of sequences. Essentially, BLAST scans each protein sequence and discards those that lack certain “words” it identified as crucial for meaningful alignments.

Although BLAST is an efficient approximation algorithm, it is limited in several ways. First, it is an exhaustive search algorithm that analyzes all sequences in a protein database. Furthermore, it only works well for sequences that are very closely related: sequences with similar lengths, similar number of domains (protein units of structural/function importance), and equivalent domain arrangements. Therefore, BLAST, and its more sensitive variant, PSI-BLAST are not good in capturing remote homologies.

Modern information retrieval systems rely extensively on indexes to provide immediate access to relevant records in large databases. As opposed to running an exhaustive search for every query, modern database systems typically start with a time consuming pre-processing step to build an index that has great amortization value over the long run. For example, so-called “inverted indexes” are used to specify all of the documents in which a particular word occurs, much like how the index section of a book specifies which pages contain certain keywords.

Protein homology searches can also be based on inverted indexes rather than exhaustive methods. One of the main contributions of this dissertation is a framework for building protein sequence databases indexed by biologically interesting and statistically significant sequence motifs. These sequence motifs can be derived from any multiple sequence alignments such as alignments from the BLOCKS database [40] and PSI-BLAST results. By utilizing sequence motifs as words, homology searches will be context-driven, potentially delivering more accurate and
sensitive results, while saving much computational time.

1.3 Statistical Analysis of Protein Motifs

Despite the time consuming nature of BLAST searches, BLAST has become the de-facto standard for many scientists. Many scientists find it very useful that BLAST measures the statistical significance of its homology results. The statistical significance is derived from well-known probability models of sequence alignment scores. Instead of computing significance of sequence alignments, we develop theories for evaluating the significance of motif matches on protein sequences. This is the main theme of Chapter 3.

One common way to model sequence motifs is by frequency matrices. A frequency matrix records the amino acid counts for positions along a multiple sequence alignment. Extensions include smoothing positional amino acid distributions by averaging [35], including pseudocounts or Dirichlet priors [39, 81], and employing sequence and position specific weightings [84]. These matrices measure how amino acids are preferred or selected against at positions along the motif, and they are generally called Position Specific Scoring Matrices (PSSMs) [35].

However, these methods do not consider data over-fitting problems. For example, if a position captures hydrophobicity and it is deeply embedded in a hydrophobic core, then it can accommodate any of the 10 hydrophobic amino acids with equal likelihood. But multiple alignments usually carry sampling noise, and other non-hydrophobic acids might be detected. Current PSSM estimation procedures will include them into the PSSM, whereas a more robust approach is to use statistical tests to filter them out.

In Chapter 3, we present two methods that generate statistically robust motif PSSMs from multiple sequence alignments. A series of biologically motivated statistical tests help determine the degree of amino acid preference or non-preference in different positions along a conserved pattern. Using these models of sequence conservation, we exhaustively score sequence fragments across large protein databases.
High scoring sequence fragments that occur much more frequently than expected are given high significance.

In the context of molecular evolution, we are using large samples to measure the conservation of critical functional/structural elements in proteins. Similar studies have been performed by Tatusov et al. [86], but our work is the first to scale up computation for large databases. In this dissertation, we present our approach to estimating the statistical significance of sequence motif indexes as well as discussing the computational challenges involved. Our motif indexing framework also naturally leverages a large amount of unannotated sequence data to improve the statistical significance and database coverage of its index.

When observed at the most elementary level, protein sequences are strings of 20 symbols. However, further analysis reveals much structure within the seeming randomness: words, syntactical and semantical constructs often exist as building blocks of proteins [80]. We discuss ways to decode crucial language structures in proteins when the location of the conservation units are known.

We define motif families in the context of co-occurrence and collocation clusters. In co-occurrence analysis, sequence motifs that commonly occur together on the same proteins form a motif group. Collocation analysis further explore syntactical and semantical structures of motifs that occur in close proximity. These results help us construct generalized templates that capture multiple motifs in a single motif model. One goal is to compress our indexes by discovering motif relationships to support faster sequence homology searches.

1.4 Motif Finding using Random Projections

In Chapters 3, we rely on existing sequence alignment libraries to study the distribution of known motifs in large sequence databases. In Chapter 4, we introduce a method for finding motifs in protein sequences. The method can be naturally extended to handle genomic sequences for discovering important gene regulatory elements such as promoters and ribosomal binding sites in prokaryotic genomes.
The problem of finding signals in biological sequences is a well-studied problem in computational biology. Essentially, motif finding is a sequence clustering problem that is further complicated by weakly conserved positions. There are two broad categories of motif finding algorithms: stochastic methods such as CONSENSUS [42], Gibbs sampler [55], and MEME [6] that are based on sequence profile models; and combinatorial methods such as TEIRESIAS [76], WINNOWER and SP-STAR [71], and MITRA [29] that are based on variations of the consensus sequence models.

In a series of papers [15, 17, 16], Buhler et al. present an effective motif discovery algorithm that uses random projections and locality sensitive hashing to aggregate similar sequence fragments. We show another method for embedding sequence fragments in Euclidean space. Our method applies rigorous statistical analysis of random projections to discover novel motifs (and motif families). We show two approaches for organizing random projections: a variation of the locality sensitive hashing algorithm [34] and the method of random projection trees (RP-Trees).

Instead of iteratively re-sampling the results of a particular seed projection (as in Buhler’s LSH-ALL-PAIRS), we re-seed each execution of our algorithm with fresh parameters and mark the detections of recurrent sequence signals in the sequence database. We explore two types of random projections for sequence fragments. The first method computes random projections for whole sequence fragments and the second method projects a random selection of positions in a fragment window. Using the random projection values, the sequence signals are subsequently grouped into families based on the distribution of projection values and distribution of detections over the sequence set.

This work is directly applied to the motif finding problem, analyzing all documented proteins in the UniProtKB database to aid protein homology search. It complements the sequence indexing framework introduced in Chapter 3. We also apply our algorithm for the analysis of microbial protein signatures. The analysis is an essential component to our approach for a new bioinformatics problem we
introduce in Chapter 5. Specifically, microbial protein signatures are used as one of many clues to discover whole or partial protein coding regions in microbial genome fragments.

1.5 Microbial Gene Finding without Assembly

An estimated 99% of microbial species are resistant to culture under standard laboratory techniques [43] and as such, little is known about one of the most important and abundant type of organism on Earth. Recent metagenomics initiatives aim to study communities of microbial organisms directly in their natural habitats, without the need for species isolation and laboratory culture [23, 37, 83]. However, the bacterial diversity in these natural environments prohibits typical shotgun sequence procedures for whole genome reassembly.

The acid mine biofilm community [88] is a relatively simple model containing only five species. Sequencing 75 Mbps generates sufficient genome coverage to reconstruct near complete genomes for two of the microbes. However, in the more recent Sargasso Sea environmental sampling project [90], sequencing 1.6 Gbps produced an estimated 1,800 species (including 148 new species) and partially complete genome assemblies were only available for three species. It is estimated that 50% of the sequences generated are single reads. Since the single reads are roughly 1,000 bps in length, and prokaryotic genes are usually short and lack introns, it is very likely that each of the single reads contains partial or whole bacterial genes.

Traditional hidden Markov model (HMM) based prokaryotic gene finders like GLIMMER [78, 22] and EasyGene [54, 66] typically assumes a particular genome model and uses Expectation Maximization (EM) unsupervised training to learn the HMM parameters specific to a microbial species. Hence these gene finders require a representative sample of whole genes, usually also including gene flanking regions, for proper understanding of each species’ gene structure. Because of this, it is unlikely that these traditional gene finders will be successful with metagenomics
sequences, where many sequences are fragmented single reads possibly originating from unknown microbial species.

In Chapter 5 of this dissertation, we present a data integration approach to discovering partial or whole genes in genome fragments originating from a mixture of bacterial species. The data is simulated by combining copies of genomic fragments extracted from currently known and complete microbial genomes. Our approach is based on the well-known boosting algorithm [31], where we perform confidence rated prediction of the presence of complete or partial genes based on the genomic features found on the fragments. In the boosting terminology, we have a series of weak learners where each encodes a particular feature that is typical of prokaryotic genes.

We separate our weak learners for signals upstream and downstream from a candidate start codon. For signals upstream from a start codon, we use weight matrices and spacer distance distributions for ribosomal binding sites, promoters, and other transcription factor binding sites as possible gene start signals. For signals downstream from a start codon, we use coding potentials based on codon usages analyzed from mixtures of bacterial coding regions as another weak learner. We also use our random projection analysis of microbial protein signatures presented in Chapter 4 and microbial protein signatures based on context trees [93, 94] as other signals typical of prokaryotic genes.

Dissertation organization. The remainder of the dissertation is organized as follows. In Chapter 2, we discuss the multi-class protein classification problem and present our approach that uses code words to encode multi-class hierarchies. In Chapter 3, we focus on the analysis and application of protein sequence motifs. Then in Chapter 4, we present alternative approaches to the motif finding problem. The algorithms presented are also applied to discovering microbial protein signatures that are prevalent across microbial species. In Chapter 5, we look at a new problem relevant to metagenomics projects and show how boosting is well suited to detecting probable coding regions in prokaryotic gene fragments.
Multi-Class Protein Classification using Adaptive Codes

Many statistical, homology-based methods have been developed for detecting protein structural classes from protein primary sequence information alone. They can be categorized into three major types of methods: pairwise sequence comparison algorithms [3, 82], generative models for protein families [52, 70], and discriminative classifiers [47, 57, 59].

Many recent studies have shown that discriminative classifiers, such as support vector machines (SVMs), outperform the other two types of protein classification methods [56] in the context of binary remote homology detection—prediction of whether a sequence belongs to a single structural class or not—especially when incorporating unlabeled protein data [91, 53]. However, it is uncertain how to combine these predictive binary classifiers properly in order to tackle the multi-class problem of classifying protein sequences into one of many structural classes.

In the machine learning literature, two main strategies have been devised to tackle multi-class problems: formulating large multi-class optimization problems that generalize well-known binary optimization problems such as support vector machines [89, 92], or reducing multi-class problems to a set of binary classification problems and processing the binary predictions in simple ways to obtain a multi-class prediction [1, 25]. The difficulty with the first method is that one usually ends up with a complex optimization problem that is computationally expensive.
The second method is more computationally tractable, since it involves training a set of binary classifiers and assigns to each test example a vector of real-valued discriminant scores or binary prediction rule scores which we call the output vector for the example.

In the standard one-vs-all approach, one trains \( N \) one-vs-the-rest classifiers to obtain a length \( N \) output vector, and one predicts the class with the largest discriminant score; standard all-vs-all is similar, but one trains all pairwise binary classifiers to obtain a length \( N(N - 1)/2 \) output vector [1]. One can also represent different classes by binary vectors or output codes in the output vector space and predict the class based on which output code is closest to the binary output vector for the example [25, 19]. (We will use the terms “output space” and “code space” interchangeably for the output vector space.) This approach, called error-correcting output codes (ECOC), appears more flexible than the other standard methods, though a recent empirical study suggests that the one-vs-the-rest approach performs well in most cases [75].

There has not been much work on applying newer multi-class techniques for protein classification, though recently, Tan et al. [85] approached the multi-class fold recognition problem by statistically integrating one-vs-all and all-vs-all classifiers at the binary rule level to generate new classifiers for the final problem. One failing of one-vs-all is that it assumes that the margins of the component binary classifiers are comparable, so that the individual classifier with the largest prediction corresponds to the best class. This assumption is often invalid in practice.

One proposed remedy for SVM classifiers in particular is to fit a sigmoid function to the predicted margins for each classifier [72]. After this procedure, the output probabilities rather than the margins are compared in one-vs-all. However, in many applications, the training data may be insufficient to fit the sigmoids accurately, or the sigmoids may be poor models for the margin distributions. Moreover, one-vs-all and the other standard output vector approaches do not take advantage of known relationships between classes, such as hierarchical relationships in the protein structural taxonomy.
We present a simple but effective multi-class method for protein fold recognition that combines the predictions of state-of-the-art one-vs-the-rest SVM protein classifiers by learning in the output space. In order to solve the problem that prediction scores from different classifiers are not on the same scale, we solve an optimization problem to learn a weighting of the real-valued binary classifiers that make up the components of the output code.

Instead of using \textit{ad hoc} output codes as in ECOC, we design codes that are directly related to the structural hierarchy of a known taxonomy, such as the manually curated Structural Classification of Proteins (SCOP) [65], with components that correspond to fold and superfamily detectors. We note that Rätsch \textit{et al.} [74] considered a more general and difficult problem of adapting codes and embeddings, that is, learning both the code vectors and the embedding of the vector of prediction scores in output space via a non-convex optimization problem. In addition, Crammer and Singer [19] formulated another more general problem of learning a mapping of all inputs to all outputs.

By restricting ourselves to the simpler problem of reweighting the output space so that our fixed codes perform well, we are able to define a convex large-margin optimization problem that can be efficiently solved with recent methods and is tractable in very large scale settings. Unlike the other two approaches, we can also choose which loss function we wish to optimize—for example, in protein classification, we can use the balanced loss, so that performance on the large classes does not dominate the results—and we can make use of hierarchical labels.

To set up the optimization problem during training, we use a cross-validation scheme to embed protein sequences in an output space, representing each protein as a vector of SVM discriminant scores. Suppose the number of superfamilies and folds in a SCOP-based data set is \( k \) and \( q \) respectively. Our approach for solving the multi-class protein classification problem involves producing a real-valued output vector \( \vec{f}(x) = (f_1(x), ..., f_{k+q}(x)) \) for each test sequence \( x \), where the \( f_i \) are binary SVM superfamily or fold detectors trained using profile string kernels [53], and using \( (k + q) \)-length code vectors \( C_j \) that encode the superfamily
and fold of a protein class as a bit vector. We use training data to learn a weight vector \( W = (W_1, \ldots, W_{k+q}) \) to perform multi-class predictions with the weighted code prediction rule,

\[
\hat{y} = \arg \max_i (W \cdot \tilde{f}(x)) \cdot C_i,
\]

where \( W \cdot \tilde{f}(x) \) denotes component-wise multiplication.

We learn \( W \) by a cross-validation set-up on the training set, using either a ranking perceptron or structured SVM algorithm. The full methodology consists of five steps:

1. Split the training data into 10 cross-validation sets;
2. Learn fold- and superfamily-level detectors from the partitioned training set—performing fold recognition and superfamily recognition on the held-out cross-validation sets, thereby generating training data for code weight
 learning;

3. Use either the ranking perceptron algorithm or the structured SVM method for learning the optimal weighting of classifiers in code space;

4. Re-train superfamily and fold detectors on the full training set;

5. Test on the final untouched test set.

Figure 2.1 shows a summary of the above steps, and Figure 2.2 shows the cross-validation scheme for learning $W$.

The rest of the chapter is organized as follows. First, we show how to incorporate structural hierarchy into codes and to combine base classifiers through code embeddings using the ranking perceptron algorithm or structured SVM. We then provide experimental results on multi-class fold recognition, comparing our approach with two alternatives: an unweighted combination of maximum margin one-vs-all base classifiers, PSI-BLAST nearest neighbor searches on the non-
redundant protein database, and Platt’s sigmoid fitting [72] for direct classifier comparisons. We also augment our original codes with PSI-BLAST code elements, obtaining significant improvement over one-vs-all and basic PSI-BLAST methods for remote homology detection at the superfamily level. The current work is an expanded version of a conference proceedings paper [45]. We have added results for the multi-class superfamily recognition problem using extended codes that include PSI-BLAST based probabilistic superfamily detectors.

2.1 Profile-based Protein Classifiers

For our base binary classifiers, we use profile-based string kernel SVMs [53] that are trained to recognize SCOP fold and superfamily classes. We call these trained SVMs fold detectors and superfamily detectors.

The profile kernel is a function that measures the similarity of two protein sequence profiles based on their representation in a high-dimensional vector space indexed by all $k$-mers ($k$-length subsequences of amino acids). Intuitively, we use each $k$-length window of the sequence profile to define a positional mutation neighborhood of $k$-mers that satisfy a likelihood threshold, and the underlying feature map counts the $k$-mers from all the positional neighborhoods.

Specifically, for a sequence $x$ and its sequence profile $P(x)$ (e.g. PSI-BLAST profile), the positional mutation neighborhood at position $j$ and with threshold $\sigma$ is defined to be the set of $k$-mers $\beta = b_1b_2\ldots b_k$ satisfying a likelihood inequality with respect to the corresponding block of the profile $P(x)$, as follows:

$$M_{(k,\sigma)}(P(x[j + 1 : j + k])) = \left\{ \beta = b_1b_2\ldots b_k : -\sum_{i=1}^{k} \log p_{j+i}(b_i) < \sigma \right\}.$$

Note that the emission probabilities, $p_{j+i}(b), i = 1\ldots k$, come from the profile $P(x)$—for notational simplicity, we do not explicitly indicate the dependence on $x$. Typically, the profiles are estimated from close homologs found in a large sequence database; however, these estimates may be too restrictive for our purposes.
Therefore, we smooth the estimates using the training set background frequencies $q(b)$, where $b \in \Sigma$ (the alphabet of amino acids), via

$$\tilde{p}_i(b) = \frac{p_i(b) + tq(b)}{1 + t},$$

for $i = 1 \ldots |x|$ and where $t$ is a smoothing parameter. We use the smoothed emission probabilities $\tilde{p}_i(b)$ in place of $p_i(b)$ in defining the mutation neighborhoods.

We now define the profile feature mapping as

$$\Phi^{\text{Profile}}_{(k,\sigma)}(P(x)) = \sum_{j=0\ldots |x|-k} (\phi_\beta(P(x[j+1:j+k])))_{\beta \in \Sigma^k},$$

where the coordinate

$$\phi_\beta(P(x[j+1:j+k])) = \begin{cases} 
1 & \text{if } \beta \in M_{(k,\sigma)}(P(x[j+1:j+k])), \\
0 & \text{otherwise.} 
\end{cases}$$

Note that the profile kernel between two protein sequences is simply defined by the inner product of feature vectors,

$$K^{\text{Profile}}_{(k,\sigma)}(P(x), P(y)) = \langle \Phi^{\text{Profile}}_{(k,\sigma)}(P(x)), \Phi^{\text{Profile}}_{(k,\sigma)}(P(y)) \rangle.$$

The use of profile-based string kernels is an example of semi-supervised learning, since unlabeled data in the form of a large sequence database is used in the discrimination problem (specifically, to estimate the probabilistic profiles). Moreover, profile kernel values can be efficiently computed in time that scales linearly with input sequence length.

### 2.2 Embedding Base Classifiers in Code Space

Suppose that we have trained $q$ fold detectors. Then for a protein sequence, $x$, we form a prediction discriminant vector, $\vec{f}(x) = (f_1(x), ..., f_q(x))$. The simple one-versus all prediction rule for multi-class fold recognition is

$$\hat{y} = \arg\max_j f_j(x).$$
The primary problem with this prediction rule is that the discriminant values produced by the different SVM classifiers are not necessarily comparable. While methods have been proposed to convert SVM discriminant scores into probabilistic outputs, for example using sigmoid fitting [72], in practice there may be insufficient data to estimate the sigmoid, or the fit may be poor.

In contrast, our approach learns optimal weighting for a set of classifiers, thereby scaling their discriminant values and making them more readily comparable. We also incorporate information available from the superfamily detectors for doing multi-class superfamily and fold recognition by designing output codes.

We construct our codes to incorporate knowledge about the known structural hierarchy provided by SCOP. Define for superfamily classes \( j \in \{1, ..., k\} \), code vectors \( C_j = (\text{superfam}_j, \text{fold}_j) \), where \( \text{superfam}_j \) and \( \text{fold}_j \) are vectors with length equal to the number of known superfamilies (\( k \)) and folds (\( q \)), and each of these two vectors has exactly one non-zero component corresponding to structural class identity. Each component in \( C_j \) is known as a code element and represents the discriminant value given by the corresponding classifier.

We adapt the coding system to fit the training data by learning a weighting of the code elements (or classifiers). To learn the weight vector \( W \), we formulate a hard margin optimization problem as

\[
\min_W ||W||_2^2,
\]

subject to

\[
(W * f(x_i)) \cdot (C_{y_i} - C_j) \geq 1, \quad \forall j \neq y_i.
\]

Intuitively, our problem is to find an optimal re-weighting of the discriminant vector elements such that a weighted embedding of the discriminant vector in code space \( \mathbb{R}^{k+q} \) will exhibit large margins to discriminate between correct and incorrect codes (i.e. class identity). Figure 2.3 shows a hypothetical re-weighting of codes that results in two different predictions. In this example, the code vectors \( C_j \) are simply unit vectors in the direction of the coordinate axes. The weighted code space would predict class 3 instead of class 1 as predicted in the unweighted case.
\[ \vec{f}(x) = (f_1(x), f_2(x), f_3(x)) \]

\[ \vec{C}_2 = (0, 1, 0) \]

\[ \vec{C}_3 = (0, 0, 1) \]

\[ \vec{C}_1 = (1, 0, 0) \]

\[ W^* \vec{f}(x) = (W_1 f_1(x), W_2 f_2(x), W_3 f_3(x)) \]

Figure 2.3: The embedding of a protein sequence discriminant vector in unweighted code space (top). The embedding of the same protein sequence discriminant vector in a weighted code space (bottom).
2.3 Learning Code Weights

We use two approaches to find approximate solutions to this optimization problem, the ranking perceptron and the structured SVM algorithm.

2.3.1 Ranking Perceptrons

The ranking perceptron algorithm [18] is a variant of the well-known perceptron linear classifier by Rosenblatt [77]. In our experiments, the ranking perceptron receives as input the discriminant vectors for training sequences and produces as output a weight vector $W$ which is a linear combination of the input vectors projected onto code space. We modify the ranking perceptron algorithm such that it will learn our weight vector, $W$, by satisfying $n$ constraints

$$W \cdot (\vec{f}(x_i) * C_{y_i} - \vec{f}(x_i) * C_{j}) \geq m, \forall j \neq y_i$$

(2.1)

where $m$ is the size of the margin we enforce (Figure 2.4). In the pseudocode, $\nu$ is the learning parameter; $k_i = |\{y_j : y_j = y_i\}|$ for balanced-loss, and $k_i = 1$, for zero-one loss. We also typically have $m = 2$.

The update rule of the ranking perceptron algorithm can be different depending on what kind of loss function one is aiming to optimize. In standard zero-one loss (or classification loss), one counts all prediction mistakes equally,

$$l_z(y, \hat{y}) = \begin{cases} 
1 & \text{if } \hat{y} \neq y; \\
0 & \text{otherwise.}
\end{cases}$$

The final zero-one empirical loss is $\frac{1}{n} \sum_i l_z(y_i, \hat{y}_i)$. In balanced loss, the cost of each mistake is inversely proportional to the true class size,

$$l_b(y, \hat{y}) = \begin{cases} 
\frac{1}{|y : y = y|} & \text{if } \hat{y} \neq y; \\
0 & \text{otherwise.}
\end{cases}$$

The final balanced empirical loss is $\frac{1}{|Y|} \sum_i l_b(y_i, \hat{y}_i)$, where $Y$ denotes the set of output labels.
(A) Code weights learning

1: Define $F(x, y) = W \cdot (\bar{f}(x) \ast C_y)$
2: Input $\nu$:
3: $W \leftarrow 0$
4: for $i = 1$ to $n$ do
5:   $j = \arg \max_p F(x_i, p)$
6:   if $F(x_i, y_i) - m < \max_{q \in \{Y - j\}} F(x_i, q)$ then
7:     $W \leftarrow W + \nu k_i^{-1} (\bar{f}(x_i) \ast C_{y_i} - \bar{f}(x_i) \ast C_j)$
8:   end if
9: end for
10: Return $W$

(B) Class prediction

1: Define $F(x, y) = W \cdot (\bar{f}(x) \ast C_y)$
2: Input $W, x_i$:
3: Return $\hat{y} \leftarrow \arg \max_j F(x_i, j)$

Figure 2.4: Pseudocode for the ranking perceptron algorithm used to learn code weighting.
2.3.2 Structured SVMs

Support vector machines have been applied to problems with interdependent and structured output spaces in Tsochantaridis et al. [87]. They make use of a combined input-output feature representation $\psi(x, y)$ as training vectors to learn a linear classification rule,

$$\hat{y} = \arg \max_{y \in Y} \langle W, \psi(x, y) \rangle.$$ 

Specifically, they use the $\psi(\cdot, \cdot)$ relation to discover input-output relations by forming $n|Y| - n$ linear constraints. These linear constraints specify that all correct input-output structures must be clearly separated from all incorrect input-output structures,

$$\langle W, \delta \psi_i(y) \rangle > 0 \quad \forall i, y \neq y_i,$$

where $\delta \psi_i(y) \equiv \psi(x_i, y_i) - \psi_i(x, y)$. By defining, $\psi(x, y) = \tilde{f}(x_i) \ast C_y$, we arrive at linear constraints that are a special case of Equation 2.1. Using standard maximum-margin methods like SVMs, we obtain the hard margin problem

$$\min_W \frac{1}{2} ||W||^2$$

$$\forall i, \forall y \in \{Y - y_i\} : \langle W, \delta \psi_i(y) \rangle \geq 1,$$

and the soft margin problem

$$\min_W, \xi \frac{1}{2} ||W||^2 + \frac{C}{n} \sum_{i=1}^{n} \xi_i$$

$$\forall i, \xi_i \geq 0; \forall i, \forall y \in \{Y - y_i\} : \langle W, \delta \psi_i(y) \rangle \geq 1 - \xi_i,$$

where the $\xi_i$ correspond to slack variables (the amount an example can violate the margin), and $C$ corresponds to the trade-off between maximizing the margin and the degree to which noisy examples are allowed to violate the margin. Intuitively, our definition of $\psi$ defines the distance between two different protein embeddings in code space, and we are using large margin SVM methods to find the relative weighting of the dimensions in code space. Moreover, one can optimize balanced loss by rescaling the slack variables $\xi_i \leftarrow \frac{\xi_i}{l_b(y_i, y)}$ in the constraint inequalities.
2.4 Data Set for Training Base Classifiers

We review the remote homology problem setup using a hypothetical SCOP hierarchy shown in Figure 2.5. In a typical superfamily recognition problem for superfamily \{1.2.1\}, we will have families \{1.2.1.1, 1.2.1.2\} as positive training data, family \{1.2.1.3\} as positive test data, superfamily \{1.2.2\} as negative training data, and superfamily \{1.2.3\} as negative test data.

Our training and testing data is derived from the SCOP 1.65 protein database. We use ASTRAL [13] to filter these sequences so that no two sequences share greater than 95% identity. Before running the experiments, we first remove all folds that contain less than three superfamilies so that there is a meaningful hierarchy in the remaining data.

In addition, we select at least one superfamily from within the fold which contains more than one family, so that a remote homology detection problem can be properly constructed—that is, we want a superfamily recognition problem where the test sequences are only remotely related to (not from the same SCOP family as) the training sequences. Note that in some cases, there can be several superfamily recognition experiments within the same fold if there are enough multi-family superfamilies. Our data filtering scheme results in a data set that contains 46 SCOP superfamilies belonging to 24 unique SCOP folds. Details on training the binary superfamily and fold detectors are given below.

2.4.1 Superfamily Detectors

We completely hold out 46 SCOP families for final testing of multi-class prediction algorithms. In order to make our classification problem hard and meaningful, we find families that comprise at most 40% of the total number of sequences in their corresponding superfamilies, leaving at least 60% for learning. The selected held-out families are completely untouched during any stages of training, so that a valid remote homology detection problem is in place. In general, our experimental setup is the same as the [47] remote homology detection experiment design, but we
also ensure that the negative data for each superfamily detector does not include positive training data of other superfamily detectors in the same fold.

2.4.2 Fold Detectors

Fold recognition setup is analogous to the superfamily recognition setup, except that we now deal with recognition one level up the hierarchy. Furthermore, the sequences used in the superfamily setup are a proper subset of those used in the fold setup.

2.4.3 Data Set for Learning Weights

As outlined in Figures 2.1 and 2.2, the first two stages of the method involve generating the appropriate training data for our weight learning algorithms. Specifically, we need vectors of SVM discriminant scores—given by trained fold and superfamily detectors—to embed sequences in output space in order to learn
the code weighting. To do this, we use the following cross-validation scheme: we generate cross-validation experiments by randomly partitioning the positive training data of each superfamily (fold) detector into 10 mutually exclusive sets; in each cross-validation experiment, we train the superfamily (fold) detector in order to produce a set of discriminant vectors (SVM predictions) on the held-out set. Then we collect all the discriminant vectors from the 10 held-out sets to use as training data for our weight learning algorithms.

2.5 Experimental Setup

2.5.1 Baseline Methods

Our main baseline comparison is the one-vs-all approach, using the one-vs-all superfamily (fold) detectors directly for multi-class superfamily (fold) recognition with the prediction rule is $\hat{y} = \arg \max_j f_j(x_i)$. We also test one-vs-all with both fold and superfamily detectors; in this case, if a superfamily detector has largest discriminant score, our prediction is the fold that the superfamily belongs to.

To compare against a standard refinement of one-vs-all, we report the error rates achieved by applying Platt’s sigmoid fitting [72] on each one-vs-all base classifier. We obtain the discriminants for learning the sigmoid function parameters by doing a 10-fold cross-validation on the training set. Furthermore, our code weight learning approach is compared with the multi class performance of the widely used PSI-BLAST algorithm. We test two variants of a nearest neighbor approach using PSI-BLAST searches on the non-redundant protein database, called the query-based profile and target-based profile approaches.

For the query-based profile approach, we build a profile around each test sequence and use the resulting profiles to rank all training sequences. We take the training sequence with the smallest E-value and predict its superfamily (fold) label for the query. For the target-based profile approach, we build a profile around each training sequence and use it to rank the list of test sequences, thereby comput-
ing an E-value for each test sequence. Then for each query, we use the training sequence that assigned the lowest E-value for our superfamily (fold) prediction.

### 2.5.2 Extended Codes

To test the versatility of our code learning method, we double the dimensionality of our original SVM-based codes with another set of PSI-BLAST-based classifiers. To construct the PSI-BLAST-based classifiers, we first use a labeled training data set of \(m\) proteins with known labels \(y_i\), and a matrix \(D\) of PSI-BLAST E-values for these proteins, where \(D_{ij} = S_j(i)\) is the E-value between protein \(i\) and \(j\) using \(j\) as the query and \(i\) as the target. We then compute a histogram of empirical frequencies for the PSI-BLAST E-values.

More specifically, we choose bin centers \(v_k\), and compute \(n_k\), the number of times \(D_{ij}\) falls into the bin centered at \(v_k\), and \(s_k\), the number of times that the latter occurs when \(i\) and \(j\) are in the same superfamily. We then compute \(s_k/n_k\), the empirical probability of belonging to the superfamily of interest for the bin. Finally, we approximate the mapping of E-values to empirical probability by linearly interpolating the estimated probabilities.

Using our E-value to empirical probability mapping, we restrict our PSI-BLAST-based classifier outputs to be in the range of [0,1]. This mapping allows us to create an additional PSI-BLAST-based superfamily (fold) classifier for each SVM-based one-vs-all superfamily (fold) classifier.

### 2.5.3 Previous Benchmark

Finally, to test our method on a previously published benchmark, we apply our code weight learning algorithm on an earlier multi-class protein fold recognition data set of Dubchak et al. [26] consisting of 27 SCOP folds, with 299 training examples and 383 test examples. More details on this data set are given in the results section.
2.6 Code Weight Learning Algorithm Setup

As base classifiers, we use the profile kernel SVM with profiles generated by PSI-BLAST on the non-redundant protein database. We perform cross-validation experiments which resulted in a total of 1772 training discriminant vectors for code weight learning. We use the SPIDER machine learning library to train our individual SVM detectors.

2.6.1 Ranking Perceptron on One-vs-all Classifier Codes

When using the ranking perceptron algorithm to learn code weights, we first randomly permute the training data set. Let $f_i$ be the vector containing the $i$th dimension element of all codes and $\nu$ be the ranking perceptron algorithm’s learning parameter, we pre-compute $\theta = \sum_i var(f_i \cdot \nu)$ for the training codes. We run the perceptron algorithm on the training data iteratively, until the squared norm between successive weight vectors is less than $\theta$. For statistical robustness, we report results that average final error rates over 100 randomized runs. The learning parameter for all ranking perceptron experiments is set to 0.01, and the required margin is chosen to be $m = 2$.

2.6.2 Structured SVM on One-vs-all Classifier Codes

We implemented an instance of SVM-Struct [87] and used it to learn code weights with the minimum constraint violation $\epsilon$ set at $10^{-8}$. The SVM margin-slack trade-off variable $C$ is set at 0.1 for all structured SVM experiments.

2.6.3 Ranking Perceptron on One-vs-all Classifier Codes with PSI-BLAST Extension

We set the initial weights on the PSI-BLAST portion of the codes to 0.1. We also use two learning parameters, 0.001 for the SVM portion, and $10^{-6}$ for the PSI-BLAST portion. This choice effectively stops our algorithm from adjusting
weights in the PSI-BLAST portion. We take this approach because the individual PSI-BLAST codes are based on empirical probabilities derived from E-values and hence should already be comparable to each other. We use the same required margin of \( m = 2 \) on the ranking perceptron algorithm. For statistical robustness, we report results that average final error rates over 100 randomized runs.

### 2.7 Results

#### 2.7.1 Fold and Superfamily Recognition Results

We report overall multi-class fold and superfamily recognition results in Table 2.1. Table 2.2 and Table 2.3 show the results of using fold-only and superfamily-only codes on the fold recognition and superfamily recognition problems respectively. Table 2.4 shows the multi-class fold recognition results partitioned into SCOP classes (the top level of the SCOP hierarchy). The ranking perceptron is trained using balanced loss. The error rate variances of reported averaged perceptron runs are in the order of \( 10^{-4} \) to \( 10^{-6} \). Note that for experiments with full-length codes, we employ a different measure of fold loss that is different from the simple “hard” rule of checking output label equality (\( \hat{y} = y \)).

In the full-length codes scenario, a prediction is considered correct even if the label belongs to a different superfamily from the correct fold, i.e. we can simply infer the fold identity from the superfamily identity given the predefined hierarchical structure of SCOP.

#### 2.7.2 Importance of Hierarchical Labels

We next investigated whether the incorporation of hierarchical structure in our codes was key to the strong improvement in accuracy for multi-class fold classification. First, we retained all fold structures intact while randomizing superfamily structures within each fold to create a new data set with true fold structures but fictitious superfamily structures. Using this new perturbed data set, we repeated
Table 2.1: Multi-Class fold and superfamily recognition error rates using \textit{full-length} and \textit{PSI-BLAST extended} codes via the ranking perceptron and an instance of SVM-Struct, compared to baseline methods.

<table>
<thead>
<tr>
<th>method</th>
<th>fold recognition</th>
<th>superfam. recognition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0/1 error</td>
<td>bal. error</td>
</tr>
<tr>
<td>sigmoid fitting</td>
<td>0.5592</td>
<td>0.7207</td>
</tr>
<tr>
<td>query-based PSI-BLAST</td>
<td>0.3728</td>
<td>0.4627</td>
</tr>
<tr>
<td>target-based PSI-BLAST</td>
<td>0.3195</td>
<td>0.4627</td>
</tr>
<tr>
<td>one-vs-all</td>
<td>0.1775</td>
<td>0.4869</td>
</tr>
<tr>
<td>SVM-Struct (0/1 loss)</td>
<td>0.1654</td>
<td>0.2998</td>
</tr>
<tr>
<td>SVM-Struct (bal. loss)</td>
<td>0.1378</td>
<td>0.3255</td>
</tr>
<tr>
<td>perceptron (0/1 loss)</td>
<td>0.1567</td>
<td>0.3095</td>
</tr>
<tr>
<td>perceptron (bal. loss)</td>
<td>0.1262</td>
<td>0.2658</td>
</tr>
<tr>
<td>hybrid (0/1 loss)</td>
<td>0.1455</td>
<td>0.3032</td>
</tr>
<tr>
<td>hybrid (bal. loss)</td>
<td>0.1213</td>
<td>0.2645</td>
</tr>
</tbody>
</table>

Table 2.2: Multi-Class fold recognition error rates. This experiment uses \textit{fold-only codes} via the ranking perceptron and an instance of SVM-Struct, compared to baseline methods.

<table>
<thead>
<tr>
<th>method</th>
<th>0/1 error</th>
<th>bal. error</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigmoid fitting</td>
<td>0.5592</td>
<td>0.7207</td>
</tr>
<tr>
<td>query-based PSI-BLAST</td>
<td>0.3728</td>
<td>0.4627</td>
</tr>
<tr>
<td>target-based PSI-BLAST</td>
<td>0.3195</td>
<td>0.4627</td>
</tr>
<tr>
<td>one-vs-all</td>
<td>0.2189</td>
<td>0.5351</td>
</tr>
<tr>
<td>SVM-Struct (0/1 loss)</td>
<td>0.1977</td>
<td>0.4726</td>
</tr>
<tr>
<td>SVM-Struct (bal. loss)</td>
<td>0.2052</td>
<td>0.5037</td>
</tr>
<tr>
<td>perceptron (0/1 loss)</td>
<td>0.1623</td>
<td>0.4054</td>
</tr>
<tr>
<td>perceptron (bal. loss)</td>
<td>0.1746</td>
<td>0.4207</td>
</tr>
</tbody>
</table>
Table 2.3: Multi-Class superfamily recognition error rates. This experiments uses superfamily-only codes via the ranking perceptron and an instance of SVM-Struct, compared to baseline methods.

<table>
<thead>
<tr>
<th>method</th>
<th>0/1 error</th>
<th>bal. error</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigmoid fitting</td>
<td>0.6830</td>
<td>0.8788</td>
</tr>
<tr>
<td>query-based PSI-BLAST</td>
<td>0.4290</td>
<td>0.4779</td>
</tr>
<tr>
<td>target-based PSI-BLAST</td>
<td>0.3964</td>
<td>0.4710</td>
</tr>
<tr>
<td>one-vs-all</td>
<td>0.2071</td>
<td>0.3807</td>
</tr>
<tr>
<td>SVM-Struct (0/1 loss)</td>
<td>0.2189</td>
<td>0.3672</td>
</tr>
<tr>
<td>SVM-Struct (bal. loss)</td>
<td>0.2071</td>
<td>0.3768</td>
</tr>
<tr>
<td>perceptron (0/1 loss)</td>
<td>0.2191</td>
<td>0.3442</td>
</tr>
<tr>
<td>perceptron (bal. loss)</td>
<td>0.2128</td>
<td>0.3322</td>
</tr>
</tbody>
</table>

Our experiments.

Our results show that the balanced error rates obtained using these randomized superfamilies (average: 0.3863) resemble the balanced error rates obtained while using just fold-only codes (average: 0.4054). Further investigation into the learned weight vectors show that our code learning algorithm abandoned most superfamily classifiers by setting their weights to 0. We conclude that the superfamily detectors and the hierarchical structure information incorporated into full-length codes play a significant role in reducing the error rates.

We also used full-length codes with supervision at the fold level only, that is, only making updates in the perceptron or enforcing constraints in SVM-Struct based on fold labels rather than superfamily labels. Full-length codes with fold-level supervision outperformed fold-only codes but did not perform as well as full-length codes with full supervision.
Table 2.4: A comparison of multi-class fold classification average accuracy using PSI-BLAST nearest neighbor methods and using ranking perceptron codes learning algorithm with full-length codes.

<table>
<thead>
<tr>
<th>SCOP class</th>
<th># ex.</th>
<th>method</th>
<th>0/1 accuracy</th>
<th>bal. accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>α (4 folds)</td>
<td>40</td>
<td>perceptron</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>query-based</td>
<td>0.8500</td>
<td>0.8155</td>
</tr>
<tr>
<td></td>
<td></td>
<td>target-based</td>
<td>0.8500</td>
<td>0.6984</td>
</tr>
<tr>
<td>β (9 folds)</td>
<td>145</td>
<td>perceptron</td>
<td>0.8562</td>
<td>0.7450</td>
</tr>
<tr>
<td></td>
<td></td>
<td>query-based</td>
<td>0.6621</td>
<td>0.6264</td>
</tr>
<tr>
<td></td>
<td></td>
<td>target-based</td>
<td>0.6483</td>
<td>0.4210</td>
</tr>
<tr>
<td>α/β (5 folds)</td>
<td>93</td>
<td>perceptron</td>
<td>0.8892</td>
<td>0.5795</td>
</tr>
<tr>
<td></td>
<td></td>
<td>query-based</td>
<td>0.5806</td>
<td>0.4516</td>
</tr>
<tr>
<td></td>
<td></td>
<td>target-based</td>
<td>0.8495</td>
<td>0.8277</td>
</tr>
<tr>
<td>α + β (3 folds)</td>
<td>18</td>
<td>perceptron</td>
<td>0.4655</td>
<td>0.4540</td>
</tr>
<tr>
<td></td>
<td></td>
<td>query-based</td>
<td>0.3889</td>
<td>0.2879</td>
</tr>
<tr>
<td></td>
<td></td>
<td>target-based</td>
<td>0.4444</td>
<td>0.3833</td>
</tr>
<tr>
<td>membrane (1 fold)</td>
<td>3</td>
<td>perceptron</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>query-based</td>
<td>0.3333</td>
<td>0.3333</td>
</tr>
<tr>
<td></td>
<td></td>
<td>target-based</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>small (2 folds)</td>
<td>39</td>
<td>perceptron</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>query-based</td>
<td>0.5128</td>
<td>0.2703</td>
</tr>
<tr>
<td></td>
<td></td>
<td>target-based</td>
<td>0.3846</td>
<td>0.4866</td>
</tr>
</tbody>
</table>
Table 2.5: Multi-Class fold recognition using codes that incorporate different taxonomic information. The ranking perceptron algorithm is configured to optimize for balanced loss.

<table>
<thead>
<tr>
<th>code type</th>
<th>bal. error</th>
</tr>
</thead>
<tbody>
<tr>
<td>full-length</td>
<td>0.2658</td>
</tr>
<tr>
<td>full-length (randomized superfamilies)</td>
<td>0.3863</td>
</tr>
<tr>
<td>full-length (supervise only on folds)</td>
<td>0.4023</td>
</tr>
<tr>
<td>fold-only</td>
<td>0.4207</td>
</tr>
</tbody>
</table>

2.7.3 Comparison with Previous Benchmark

Finally, we compared our method to previously published multi-class fold recognition results by Dubchak et al. [26]. They represent protein sequences as vectors of physicochemical features, and use standard one-vs-all and all-vs-all neural nets and SVMs for multi-class discrimination.

Since the original sequences were not retained, we extracted the current (possibly somewhat different) sequences from PDB corresponding to the given PDB identifiers. Relative to SCOP 1.65, 17% of the examples have mislabeled folds, but we keep the original labels for comparison purposes. We show in Table 2.6 that our base classification system strongly outperforms the previously published multi-class results, and despite the presence of mislabeled data, our code weight learning algorithm still manages to improve balanced accuracy by another 1%.

2.8 Discussion

We have presented a novel method for performing multi-class protein classification that significantly improves on directly using an ensemble of one-vs-all classifiers. We observe an average 50% reduction in overall balanced loss for the fold recognition problem compared to PSI-BLAST nearest neighbor methods. Both algorithms presented in Sections 2.3.1 and 2.3.2 have similar performance, support-
Table 2.6: **Application of code weight learning algorithm on benchmark data set.** SVM-2 is the previous benchmark’s best classifier. The ranking perceptron is trained using balanced loss.

<table>
<thead>
<tr>
<th>method</th>
<th>bal. accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM-2</td>
<td>43.5%</td>
</tr>
<tr>
<td>one-vs-all (profile kernel)</td>
<td>72.5%</td>
</tr>
<tr>
<td>perceptron (profile kernel)</td>
<td>73.5%</td>
</tr>
</tbody>
</table>

ing the core idea of learning weighted codes that separates individual one-vs-all classifiers.

We also extend the codes by adding PSI-BLAST classifiers and obtain significant error rate improvement for the multi-class superfamily recognition problem. Protein classification data is usually highly unbalanced, and a good learning algorithm must account for this imbalance in order to succeed. Both variants of the code learning algorithm use asymmetrical updates of constraint violations according to the observed relative class sizes. This approach strongly penalizes errors made in smaller sample populations, forcing the algorithm to concentrate on the less represented examples.

We found that combining information from different levels of the label hierarchy in multi-class protein classification yields improved performance. A whole host of recent work has focused on finding good representations for the inputs in the protein classification problem, in particular in the form of novel string kernels for protein sequence data. In this work, we focused on the complementary problem of finding a good feature representation for the outputs. This improved representation is possible because the outputs have known structure in terms of taxonomic information. We find this relevant structure by learning weights in the output space to weight detectors at different levels in the label hierarchy. We conclude that our presented approach is general and can be potentially applied to any problem with structure in the outputs.
Acknowledgments

The text of this chapter, in part, is a reprint of the material as it appears in the Proceedings of the 22nd International Conference on Machine Learning [45]. The dissertation author was the primary author, and the co-authors listed in this publication supervised the research.
Statistical Analysis of Protein Motifs: Application to Homology Search

At the time of writing, there are about 3 million sequences in the UniProtKB data set [7]. Out of these, only about 7% of the sequences are documented with extensive biological annotation in Swiss-Prot [12], and only 1% of the sequences have an associated 3D structure in the Protein Data Bank (PDB) [10]. In other words, we now know the exact amino acid sequence of many proteins, but for most of these proteins we know nothing about their structure or their function.

The Smith-Waterman (SW) score [82] is a well-known sequence similarity measure. Despite the quadratic time complexity for computing the SW alignment score, the SW score has been central to many sequence-based homology search methods. As the size of sequence data sets grow into the millions, the time complexity of SW-based scoring methods becomes a serious bottleneck. Subsequently, BLAST, PSI-BLAST, and other BLAST-like methods [3, 4, 60, 97] are developed to relieve the computational complexity of SW computation essentially through approximations.

While BLAST is an efficient approximation algorithm to the optimal SW alignment, it is extremely slow when compared to search engines such as Google which indexes billions of text documents. BLAST compares the query sequence to each and every sequence in a selected data set. In computer science, this is called exhaustive search. PSI-BLAST refines BLAST by constructing a profile after each
iteration to improve search sensitivity, but the underlying mechanisms are equivalent and the computational cost is higher.

For a database with 3 million sequences, a PSI-BLAST search with 6 iterations (and all other default parameters) takes approximately 11.5 minutes per sequence on a 2 GHz single core machine with 2 Gb physical memory. Using our method, an exhaustive index search takes approximately 5 seconds to retrieve an average of 70% of the sequences returned by PSI-BLAST (along with other sequences that are not detected by PSI-BLAST). With the current efforts in whole genome sequencing, the number of known protein sequences is quickly reaching a level that would make exhaustive search prohibitively slow. As far as we know, the most recent work on improving the speed of BLAST is based on clustering sequence databases into smaller sets, effectively reducing the search space required for processing a query sequence [46].

We propose an index-based search using amino acid patterns as indexes. These amino acid patterns are represented by position specific scoring matrices (PSSMs) [35]. Indexes are used in database systems as a fast way for accessing all records with a particular property. For example, so-called “inverted indexes” are used to specify all of the documents in which a particular word occurs. Using an inverted index, the user can quickly find all documents that contain all words in a given set.

The BLAST-like alignment tool (BLAT) by Kent [49] approached the problem with a similar solution by using an index of non-overlapping $k$-length sequence patterns ($k$-mers) over a species’ genome before matching $k$-mers found on a query sequence. BLAT can perform protein homology searches 50 times faster than BLAST, but BLAT fails to compete with BLAST in remote protein homolog searches. BLAT indices are based on perfect matches of short fixed-length $k$-mers (default length 5). Comparing the query sequence to a remote homolog, it is quite likely to have a mutation in each $k$-mer, making it impossible to find the match through the index.

What we need is to replace the $k$-mers of BLAT with a different type of pat-
tern, which would be less sensitive to common mutations and thus able to find the
commonalities between remote homologs. We base our choice on what we learned
from researchers in phylogenomics. We learned that when protein sequences are
aligned, it is usually the case that there are segments of 30–80 consecutive residues
that can be aligned well, interspersed with “filler” regions that do not align. Bi-
ologists commonly refer to these segments as “highly conserved regions”, because
the prevailing understanding is that they correspond to parts of the protein se-
quence that are important for its structure or function and therefore are conserved
through evolution.

However, even highly conserved regions do not preserve their residues exactly.
According to the neutral theory of molecular evolution [50], most mutations do
not change fitness. This is probably true also for highly conserved regions in the
following way: For most locations along the preserved sequence, there are several
alternative residues that provide the required functionality. Thus even if most
point mutations degrade the fitness of the protein, there can still be exponentially
many variants of the conserved region that have equivalent fitness. Within this
large set of variations we expect to see variations due to neutral genetic drift.

In BLAST, such variations are taken into account in a simple way. BLAST uses
a single substitution matrix that captures the basic relationships between different
amino acids. This matrix was derived using set(s) of related sequences [21, 41].
By assuming a single substitution matrix for sequence similarity scoring, BLAST
effectively ignores the effect of context. In other words, while molecular selection
theory predicts that the set of equally fit substitutions for a particular location
depends on neighboring residues, BLAST ignores that dependence by using a single
substitution matrix for all sequence locations. Following in line with this argument,
PSI-BLAST’s major improvement in terms of sequence search sensitivity is greatly
attributed to the profiles that are generated after each iteration of the algorithm.
These profiles provides the correct context for future iterations.

There are two ways to represent highly conserved regions: as a model (a PSSM),
or as a list of sequence fragments. These two representations are complementary
and we use them both. In fact, our process of pattern refinement is based on alternating between the two representations. This is very reminiscent of the Expectation Maximization method [24], which is used extensively in statistics and machine learning. In the **scanning step** we map each PSSM to a set of sequence fragments and in the **refinement step** we map the sets of sequence fragments to a new, and hopefully better, set of PSSMs.

The scanning step is conceptually simple: for a given multiple sequence alignment, we construct a PSSM and calculate the score it gives for each location in a protein database. We then find an appropriate threshold for the PSSM so that all scores above the threshold have high statistical significance. A location in which the score is higher than the threshold is a **detection**. We store the location of all detections in a database. This database forms the index which enables us to perform fast searches.

The refinement step is more involved. Here we take all the detections and estimate a new, and hopefully better, set of PSSMs. There are two parts to the refinement: estimation and clustering. Estimation is a necessary step: here we take a set of multiply aligned sequence fragments and generate a PSSM which will be a good detector for these conserved patterns. Clustering is a more elaborate step, which is not always necessary, but can lead to great improvements. The goal of clustering is to identify different patterns that are slight variants of each other and combine them into a single PSSM, thus increasing coverage.

The rest of the chapter is organized as follows. We first discuss some related work and then present our methodology for analyzing pattern significance and how it is used in the scanning and detection phase. This is followed by our algorithms for pattern refinement and PSSM construction from multiple alignments. Finally, we present the underlying components of our BioSpike search engine, showing the methods and results of evaluating protein queries and ranking protein homologs.
3.1 Related Work

One of the main motivations of this work is the need to speed up BLAST or PSI-BLAST searches while maintaining a similar level of accuracy. To this end, we choose to represent conserved amino acid patterns using PSSMs as they are compact and also sufficient in modeling statistically significant short gaps.

Our method is significantly different from BLAT as we allow a more flexible pattern index in the form of varying length PSSMs. Furthermore, we also removed the restriction for indexes to be non-overlapping. Currently with the limited set of clustered and derived motif patterns from the BLOCKS database, we are able to achieve approximately 70% coverage of UniProtKB sequences. Coverage results can be further improved by incorporating motifs derived from PSI-BLAST alignments, or motifs from other structural and functional motif libraries. The generality of the underlying statistical framework for analyzing any sequence alignments makes such extensions simple to add.

Much of the work on modeling protein family sequences is based on Hidden Markov models (HMMs) [73, 8, 14, 38, 52, 27]. One of the main advantages of HMMs is its flexible modeling capability, where one can capture variable length amino acid insertion and deletion patterns within the general protein signature. However, computing the score of a sequence with respect to an HMM requires much more computation than calculating the score for a PSSM. We believe that, in fact, much of the flexibility of HMMs is unnecessary.

Typical protein sequences consist of alternating highly conserved regions and unconserved or “filler” regions. Insertions and deletions in the conserved regions are rare and short and can be effectively ignored. On the other hand, filler regions have little statistical consistency and therefore can be treated as a concatenation of unconstrained or “wild-card” amino acids. Our exploratory analysis of multiple sequence alignments from the well-known Pfam database [9] suggests that the above description is quite accurate even though the alignment was done using HMMs. Our conclusion is that using a much simpler statistical model which consists of
free-floating PSSMs can capture the same statistical information as HMMs at a fraction of the computational cost.

In some respects, our work is very similar to the Conserved Domain Database (CDD) [62], Conserved Domain Architecture Retrieval Tool (CDART) [33], and Conserved Domain Search (CD-Search) [61] programs. These programs process query protein sequences with PSSMs using Reverse Position Specific-BLAST (RPS-BLAST) and then do various search related tasks based on the matched PSSMs.

We differ from these methods in several ways. Most importantly, perhaps, is that we use the statistics of uncurated protein sequences to refine our motifs. The number of uncurated sequences is much larger than the curated ones, and is likely to continue to grow rapidly in coming years. By using these much larger data sets we can generate PSSMs whose coverage and significance are much higher than those of CDD. Our work is also reminiscent of the research by Tatusov et al. [86], but we are the first to scale up computation for large databases and we also include robust statistical analysis of the motifs.

As we are measuring the significance of our motifs using uncurated sequences it seems possible that some of the PSSMs that we find have high statistical significance but no biological significance. While this is certainly possible, it seems to us very unlikely. That is because the statistical significance and the prevalence of the protein segments in the database (assuming the database has been screened for replicates) has to have an evolutionary explanation, i.e. an explanation in terms of fitness. This fitness might not be the fitness of the host organism, it might correspond to some parasitic life form such as a virus. Nevertheless, in order to be preserved as a protein code it seems necessary that the sequence segment manifests itself as part of an actual protein not too infrequently and that this manifestation does, in some way, contribute to the proliferation of the sequence segment.

In terms of a PSSM match’s statistical significance, CDD, CDART, and CD-Search rely on the local alignment E-value scores generated by RPS-BLAST and IMPALA [79]. While IMPALA’s method for evaluating statistical significance is based on estimating empirical parameters for the extreme value distribution
our method builds the statistical model by a comparison between the null distribution of scores (similar to eMATRIX [95]) and the empirical distribution of the scores (similar to BLOCKS). In this way, our method is more general and is not restricted to any single optimal SW local alignment.

3.2 Statistical Significance of Conserved Patterns

PSSMs are generalized probabilistic models that have been used in many successful applications as models to encode rich representation of local amino acid environments [53, 96]. They are derived from amino acid frequencies observed at specific locations over a sequence set. We describe a methodology for analyzing the statistical significance of PSSM matches on sequence locations.

When observing over a large sequence data set, we define a PSSM to be interesting and useful when it assigns high scores only to a small fraction of the possible amino acid positions (high significance) and it covers a sufficiently large set of sequences (high coverage). These two properties ensure that the PSSMs we use are non-redundant and they can be used to create a useful index to address most sequences in the data set.

We denote a PSSM as a $20 \times k$ matrix $P$ with rows corresponding to the twenty possible amino acids, and columns representing the $k$ positions along the pattern. Given a protein sequence $x = x_1 x_2 ... x_n$, we define the score for position $1 \leq i \leq n - k + 1$ as $s_i = \sum_{j=0}^{k-1} P_{x_{i+j}, j}$. In this work we restrict the PSSM entries to be in the set of integers $\mathbb{Z}$, where $P_{ij} > 0$ corresponds to a preference for amino acid $i$ in position $j$, $P_{ij} < 0$ corresponds to a preference against amino acid $i$ and $P_{ij} = 0$ corresponds to no preference. The magnitude of the entry determines the degree.

For a given PSSM $P$, we define two probability distributions over scores: the null distribution and the empirical distribution. The null distribution, denoted $\tilde{p}$, is the distribution of scores that results from drawing the amino acids $x_1 ... x_k$ independently at random according to their background probabilities (i.e. the
frequency of amino acids over the entire dataset. As the PSSM consists of only integers, the null score distribution for a given PSSM can be computed in time $O(k)$ using convolutions. The *empirical* distribution $\hat{p}$ is the distribution of scores over the complete UniProtKB database. Exact calculation of this distribution requires scanning all locations in all proteins in the database. However, sufficiently accurate estimates of $\hat{p}$ can be computed by randomly sampling the protein sequences.

We define the significance of a PSSM as the ratio between the number of locations that receive a high score to the number of such locations predicted by the null distribution. We define the significance for score $s$ as

$$\theta_s = \log \left( \frac{\hat{p}(S \geq s)}{\tilde{p}(S \geq s)} \right).$$

See Figure 3.1 for a typical example of the relationship between $\hat{p}(S \geq s)$ and $\tilde{p}(S \geq s)$.

After establishing statistical significance models for all patterns available, we detect occurrences of patterns by setting an arbitrary universal threshold $\theta > 0$. For pattern $m$, the critical score for detection is,

$$S_{m,\theta} = \arg \min_s \left[ \log \left( \frac{\hat{p}(S \geq s)}{\tilde{p}(S \geq s)} \right) > \theta \right].$$

Regions in proteins that score above $S_{m,\theta}$ are statistically significant locations for pattern $m$. We will analyze these locations in Section 3.4 to form pattern families.

### 3.3 Pattern Refinement

Given a set of aligned segments, all corresponding to the same pattern, or more generally, from the same pattern family, we present several algorithms for modeling these segments as a PSSM detector. Our goal is to design a PSSM which assigns to the given segments scores that are much higher than the scores that are expected for the null score distribution for the same PSSM. Recall that the null distribution assumes that the residues in the pattern are independent of each other. Thus making the scores for the given set of segments much higher than
Figure 3.1: The tails of the estimated null and empirical score inverse cumulative distributions for the PSSM corresponding to the RuvA BLOCK motif in Bacterial DNA recombination protein (InterPro BLOCK Id:IPB000085A).
expected from the null distribution implies that the PSSM has captured strong dependencies between the residues.

We start by computing a frequency matrix which defines the empirical distributions of amino acid in each location of the alignment. We then translate the frequency matrix into a PSSM, independently for each position, using a process of iterative model refinement. The algorithms we present can easily incorporate other existing frequency matrix estimation improvements, such as pseudocounts or Dirichlet priors [39, 81], and sequence and position specific reweightings [84]. We focus on how to transform frequency matrices into statistically robust models to capture important conservation patterns based on background amino acid probabilities.

3.3.1 Estimating PSSMs Using Mixture Models

Each column in the frequency matrix is modeled as a mixture of the background amino acid distribution and several $\delta$-distributions. We define a $\delta$-distribution as a probability distribution concentrated on one of the twenty possible amino acids. Denote the background probability and $\delta$-distribution of amino acid $i$ as $q_i$ and $\delta_i$, respectively. We model the acid’s empirical probability as,

$$\tilde{h}_i = \pi_0 q_i + \pi_i \delta_i,$$

where $\pi_0$ and $\pi_i$ are mixture ratios. By modeling the frequency column as a mixture, we deduce which amino acids are essential to a position in the pattern.

We build the mixture model iteratively, starting with a pure background distribution and adding $\delta$-distributions in a greedy fashion until the model is sufficiently similar to the empirical distribution and that the remaining discrepancy can be attributed to sampling error. As the number of sequence fragments grows, the discrepancy that is attributable to sampling error becomes smaller, which enables us to reliably infer the essentiality of more amino acids.

The KL divergence is a well known dis-similarity measure between two probability distributions, and it is defined as $D_{KL}(h||\tilde{h}) = \sum_i h_i \log(h_i/\tilde{h}_i)$. Our algorithm
first approximates the distribution $h$ using the background distribution $q$. Then it proceeds by iteratively adding $\delta$-distributions to amino acids that minimizes the KL divergence between our approximation and the true empirical distribution. The algorithm is terminated by a threshold inversely proportional to the number of sequence fragment windows, $n$. In this way, we place more confidence in refinements derived from more sequence fragments. See Figure 3.2 for the details of our algorithm. The parameter $c$ is tuned so that the average number of $\delta$-distributions per column is approximately 5.

For each PSSM column, we use our algorithm to compute the set of $\delta$-distributions, $\Delta$, and map the estimated empirical model, $\tilde{h}$, into an integer column, $z$, using the following rules,

$$z_i = \begin{cases} 
1 & \text{if } h_i > q_i \text{ and } i \in \Delta, \\
-1 & \text{if } h_i < q_i \text{ and } i \in \Delta, \\
0 & \text{otherwise}.
\end{cases} \quad (3.1)$$

3.3.2 Estimating PSSMs Using Statistical Tests

In the previous section, we discussed how to estimate PSSMs from frequency matrices by estimating a mixture model on each column of the frequency matrix. There are several improvements we will make to the previous algorithm.

First of all, the PSSM entries are restricted to the integer set \{0, \pm 1\}. This is sufficient for most BLOCKS alignments when the frequency matrices account for only 10–500 sequences. However when we refine patterns using large amounts of uncurated data, we often deal with frequency matrices that include over thousands of sequences. In these scenarios, we frequently observe that some locations in the alignment are clearly more conserved than others. To model this properly, we need to use a larger set of integer values to discretize our frequency matrices.

Furthermore, in line 5 of the algorithm illustrated in Figure 3.2, the parameter $n$ is used repeatedly even after an amino acid is included in $\Delta$. This makes inaccurate calculations for the significance threshold on KL divergence differences. For
1: Input c, n:
2: Δ = {}
3: i ← 0
4: ˜h^i ← q
5: while $D_{KL}(h||\tilde{h}^i) > c/n$ do
6:     for each amino acid j do
7:         ˜h^i' ← ˜h^i
8:         ˜h^i'_j ← h_j
9:         Δ' ← Δ ∪ \{j\}
10:     Re-normalize non-Δ' elements in ˜h^i'
11:     d_j ← $D_{KL}(h||\tilde{h}^i') - D_{KL}(h||\tilde{h}^i)$
12: end for
13: k ← arg max_j d_j
14: ˜h^{i+1} ← ˜h^i
15: ˜h^i+1_k ← h_k
16: Δ ← Δ ∪ \{k\}
17: i ← i + 1
18: end while
19: Return Δ, ˜h^i

Figure 3.2: Algorithm for computing the estimated empirical model for a PSSM column and the list of amino acid with δ-distributions in the model.
example, if the first significant amino acid appears in 90% of the aligned sequences, then all others combined are 10% of \( n \). Since this is much smaller than \( n \) itself, the KL divergence differences that are accepted as significant after re-normalization are too small.

Finally, we are not taking any advantage of the fact that certain amino acids are similar and so are more likely to replace each other. For example, hydrophobic amino acids are usually exchangeable.

**Exploiting Known Relationships Between Amino Acids**

We illustrate one possible partitioning of the 20 amino acids into subsets in Figure 3.3. The graph helps us construct a series of statistical tests based on different amino acid groupings that will reveal the type of conservation in each column of a frequency matrix.

The following algorithm computes the PSSM values from frequency matrices with integer values that are correlated with the positional empirical probabilities and are validated by \( \chi^2 \) tests. Assume that the background probability of an amino acid is \( q \), and we want our PSSM values to be in the range \([-k,k]\). We will divide the \([0,1]\) probability segment the following way:

\[
\begin{align*}
(qc^{-1}, qc^1) & \rightarrow 0 \\
[qc, qc^2) & \rightarrow 1 \\
[qc^2, qc^3) & \rightarrow 2 \\
& \ldots \\
[qc^k, qc^{k+1}) & \rightarrow k \\
(qc^{-2}, qc^{-1}) & \rightarrow -1 \\
(qc^{-3}, qc^{-2}) & \rightarrow -2 \\
& \ldots \\
(qc^{-k-1}, qc^{-k}] & \rightarrow -k
\end{align*}
\]
Figure 3.3: Relationships between the amino acids.
The above mapping determines what integer values we assign to amino acids when its true probability falls within a specific range. Unfortunately, it is impossible to measure the true probability and we can only approximate it with the empirical probability derived from a multiple alignment. When the sample size is small, the neighborhood around the empirical probability for which the true probability may lie is large. The opposite is true when the sample size is large. Hence depending on the sample size, the true probability may lie one of several intervals around the empirical probability.

To determine which integer value to assign given an empirical probability, we will perform $\chi^2$ tests to evaluate the significance of the empirical probability when a specific null model (interval) is chosen. The value $c$ may be calculated as follows. Assume we want the PSSM values to be in the range $[-10, 10]$. Then, $qc^{11} = 1$. We evaluate $c = (1/q)^{\frac{1}{11}}$. If the background probability for each amino acid is the same, then $c \approx 1.3130$. The more general form of $c$ that we use is to assume each amino acid contribute $1/20$ to the background probability, and hence $c = (20/|B_k|)^{\frac{1}{20}}$. Note also that if any amino acid with empirical probability $f < qc^{-k-1}$, we automatically assign it with the smallest integer value $-k$.

The high level loop of our algorithm goes through a telescoping sequence of amino acid subsets, where each subset $B$ is a particular grouping of amino acids illustrated in Figure 3.3 ($B_1, B_2, \ldots, B_{10}$). We also include subsets that contain single amino acids (singletons). In each loop, we calculate the hypothetical PSSM entries for amino acids in $B$ by testing the statistical significance of $B$’s empirical probability with respect to its background probability. Assume $F$ is the empirical counts for amino acids in a profile and $Q$ is the amino acid background probabilities. The $\chi^2$ test statistic is

$$
\lambda = 2 \left( f' \log \left( \frac{f}{qc'} \right) + \hat{f}' \log \left( \frac{1 - f}{1 - qc'} \right) \right),
$$
where

\[
\begin{align*}
 f' &= \sum_{i \in B} F_i \\
 \hat{f}' &= \sum_j F_j - f' \\
 f &= \frac{f'}{\sum_j F_j} \\
 q &= \frac{\sum_{i \in B} Q_i}{\sum_j Q_j}.
\end{align*}
\]

The basic idea of assigning PSSM values is to think of it as a two step process. In the first step, we use a series of statistical tests to decide on a value the amino acids in grouping \( B \) can assume. The statistical tests are one sided depending on whether the empirical probability of \( B \) is higher (or lower) than its background probability. This means our statistical tests progress by trying higher (or lower) values of \( r \), i.e. placing the null hypotheses at higher (or lower) segments on the \([0, 1]\) interval. The search for \( r \) terminates when the null hypothesis cannot be rejected. Once this happens, it is uncertain if the empirical probability is distinguishable from the hypothetical background probability and all amino acids in \( B \) can assume \( r \) as a possible magnitude of conservation.

Based on the different ways we group amino acids, the PSSM value computed for an amino acid can be quite different. Our second step uses heuristics to decide what value an amino acid should take. If all non-zero values share the same sign, then the PSSM value the amino acid should take is the one with the maximum absolute value. However when signs disagree, we look at the sign of the value given by the smallest set \( B \), i.e. usually the singleton containing only that particular amino acid. Then we take the computed entry with the maximum absolute value that agrees with this sign.
3.4 Defining Motif Families

The Haemagluttinin motif (InterPro Id: IPR008635) and the Hep_Hag motif (InterPro Id: IPR008640) are observed to co-occur on bacterial haemagglutinins and invasins with high probability. We now show a framework for finding groups of biological patterns that co-occur and co-locate with high probability.

Using the pattern detections over the UniProtKB data set, we perform pattern co-occurrence and co-location analysis to define rough templates that capture sets of highly specific patterns. Our goal is to discover significant rough patterns with high coverage. These rough patterns will allow us to partition the search space, enabling fast and directed sequence searches.

3.4.1 Motifs and Zones

We analyze pattern similarity by looking at the locations of pattern detection. Before computing the pattern similarity matrices for clustering, we need to partition the set of patterns into two classes, which we call zones and motifs.

Zones are patterns which are usually detected at several consecutive positions along a sequence. For example, the polyglutamine tracts found in proteins of *Dictyostelium discoideum* are examples of zones[28]. On the other hand, motifs are highly specific patterns which are usually detected at isolated well defined locations. Motifs are the more interesting patterns as they define a highly specific conserved region which are likely to correspond to a specific 3D structure in the protein. In contrast, zones define regions in which there is an abundance of some types of amino acids. While zones have clear statistical significance, their biological significance is unclear.

We can partition the set of patterns into the two classes by analyzing the set of detections for each pattern. A motif is characterized by having a clear peak score location on the protein sequences, whereas a zone will have multiple peak score locations situated in close proximity.
3.4.2 Similarity Matrices for Sequence Motifs

The co-occurrence matrix $O$ describes how likely a pattern $m_i$ is detected with another pattern $m_j$ on a sequence $x$. The elements of the co-occurrence matrix $O$ are defined as

$$o_{i,j} = \frac{\log \left( \frac{P(m_i, m_j)}{P(m_i)P(m_j)} \right)}{\log \left( \frac{1}{\max\{P(m_i), P(m_j)\}} \right)},$$

(3.2)

where $P(m_i)$ is the fraction of sequences that contain an occurrence of the pattern $m_i$ and $P(m_i, m_j)$ is the fraction of sequences that contain occurrences of both $m_i$ and $m_j$. The range of $o_{i,j}$ is $(-\infty, 1]$, where $o_{i,j} = 0$ if $m_i$ and $m_j$ are independent, and $o_{i,j} = 1$ if one of the patterns dominates the other, i.e. pattern $m_i$ always occurs when pattern $m_j$ occurs, or vice versa. Getting $o_{i,j} < 0$ means that the $m_i$ and $m_j$ are anti-correlated. In this work we ignore information about anti-correlations.

Similarly to $O$, we compute the co-location matrix $L$ using the same formula but redefining $P(m_i)$ to be the fraction of sequence locations in which $m_i$ is detected and $P(m_i, m_j)$ to be the fraction of detections of $m_i$ ($m_j$) such that a detection of $m_j$ ($m_i$) occurs in a close proximity, i.e. the two patterns either overlap or are within a distance of at most $k$ from each other. In our experiments, we set $k = 10$.

3.4.3 Clustering Motifs

We cluster the set of motifs in two steps. Clusters found using $O$ are re-clustered using matrix $L$. In this way, the clustering algorithm at the second stage exclusively analyzes co-occurring motifs. Any clustering algorithm can be applied on matrices $O$ and $L$. We use a simple clustering scheme which first removes entries in the matrices $O$ and $L$ based on some positive thresholds $\theta_O$ and $\theta_L$. Then, we interpret each of these matrices as a graph connectivity matrix with nodes representing sequence motifs and edges representing the strength of the relation (co-occurrence or co-location score). By computing the reachability of each node on either graphs, we can group protein motifs into clusters. To prevent having weak links form large clusters of loosely connected sequence motifs, we also
require that each motif cluster satisfy a certain tightness threshold. This tightness threshold is simply the average value over the cluster’s sub-matrix in \( O \) or \( L \).

### 3.4.4 Motif Templates

We partition clusters into three types: singleton, fixed gapped, and variably gapped. Singleton clusters are motifs that have no observed relationships with any other motifs. As the name suggests, fixed gapped clusters contain motifs that co-locate at a specific gap from each other. For variably gapped clusters, motifs are observed to co-locate at varying gaps, possibly as a result of residue insertions/deletions or as a result of cyclical patterns.

PSSMs in singleton clusters are refined using the new counts but their length is maintained. PSSMs in gapped clusters are combined into “templates” which are then used to estimate new and longer PSSMs.

#### Analyzing Common Gaps Between Protein Motifs.

We define a template to be the minimal consensus window that can be used to arrange motifs from a cluster in frequently observed relative positions. Denote a cluster of motifs as \( C \). A template is the set of (motif, location)-pairs \( T = \{(m_i, l_{i,j})\} \) where \( m_i \in C \), and \( l_{i,j} \geq 0 \) is the \( j \)th location of pattern \( m_i \) in the template. Note that \(|T| \geq |C|\) for variable gapped clusters because a motif can occur several times in the template. If \( C \) is a fixed gapped cluster, then \(|T| = |C|\).

A histogram of gaps is computed for all pairs of motifs in the cluster. Denote a gap as \( g \), and its associated count as \( c \). For template construction, we aggregate the histograms into a single set forming, \( H = \{(m_a, m_b, g, c) \mid m_a \in C, m_b \in C\} \).

We use a greedy algorithm to construct the template. The algorithm iteratively incorporates pairwise motif relationships onto the template with the most frequently observed relationships first. The algorithm stops when the fraction of unsatisfied constraints drops below a specified threshold. Adding a single relationship to the template can result in adding several new (motif, location)-pairs.
1: Define $|C| = n$, $D = \{m_0\}$, $\mathcal{P} = \{m_1, \ldots, m_{n-1}\}$
2: Denote $\mathcal{H}_{a,b}$ as the set of histogram values $(m_a, m_b, g, c)$ where $m_a \in D$ and $m_b \in \mathcal{P}$.
3: Input $\mathcal{H}$, $C$:
4: $T \leftarrow \{(m_0, 0)\}$
5: Sort $\mathcal{H}$ by $c$ in descending order
6: for $i = 1$ to $n - 1$ do
7:  $w \leftarrow 0$
8:  while $w < \theta$ do
9:  $(m_a, m_b, g, c) \leftarrow \max_c \mathcal{H}_{a,b}$.
10:  Add $m_b$ to $T$ adjusting all affected relative positions based on $g$.
11:  $w \leftarrow w + \frac{c}{\sum_{\mathcal{H}_{a,b}^c}}$
12:  $H_{a,b} \leftarrow (m_a, m_b, g, c)$.
13: end while
14: $D \leftarrow D + m_i$.
15: $\mathcal{P} \leftarrow \mathcal{P} - m_i$.
16: end for
17: Return $T$

Figure 3.4: Greedy algorithm for constructing template $T$.

if the template has variable gaps, as one of the motifs in the relationship might already occur in several locations in the template. Figure 3.4 outlines the basic algorithm. The threshold $\theta$ controls the flexibility (and complexity) of template $T$. When $\theta = 1$, our template $T$ will capture all observed gaps between members of the cluster $C$. The expression $\sum_g \mathcal{H}_{a,b}$ denotes the sum of all gap frequencies between motifs $a$ and $b$. 
Constructing Generalized PSSMs for Pattern Families.

After constructing a template $T$, we fit sequence fragments from motif detection sites onto the template consensus window. Since a sequence $x$ can have several fragments that fit onto the template window, a fractional weight is assigned to each fragment to avoid over-representation of a single sequence. The multiply aligned sequence fragments are then used to generate a PSSM for the motif family using methods described in Section 3.3.

Finally, we scan the template PSSMs over the entire sequence data set for another round of pattern detection. Figure 3.5 shows a comparison in significance of detections and UniProtKB database coverage between the original BLOCKS derived profiles and the refined template profiles. Coverage of the UniProtKB data set is increased, the significance of detections is also increased, and we reduced the number of patterns needed to index the data set. We can run additional refinement steps. However, the current version of the BioSpike index is based on a single refinement.

3.4.5 Zone Templates

As zones are detected at consecutive locations along a sequence, it is not possible to analyze co-location or co-occurrence of zoning patterns using the methods described for motifs. Instead of marking each location of zone detections individually, we define a center and radius for zone detections on protein sequences.

After mapping the centers and radii for each zone’s detections on sequences, we construct an amino acid probability distribution $h$ by observing the amino acid frequencies on zone windows. The amino acid distribution for a zone will be very different from the background probability distribution of amino acids over all sequences. For example, the probability of glutamine in zones that capture polyglutamine tracts will be much higher than the background probability of glutamine.

After constructing an amino acid distribution for each zone, we use either algorithms described in Section 3.3 to translate it into integer columns, $z$. We
Figure 3.5: A comparison of significance of detections and UniProtKB protein coverage between BLOCKS derived profiles and refined template profiles.
then cluster the set of $z$, grouping them based on agreements on the $z_i$ elements. Zones that have a disagreement in preference/non-preference of a particular amino acid is separated. Finally, we construct a template for each cluster of zones by computing another integer column, $z'$, using the aggregated amino acid probability distribution for zones in the cluster. Then by repeating columns $z'$ over the average length of zone PSSMs in the cluster, we define a generalized zone template PSSM.

### 3.5 Sequence Search Engine

#### 3.5.1 Building BioSpike

In general, the search engine development iterates over PSSM refinement, sequence database scan, pattern detection, and pattern clustering, in which the pattern detection stage builds the database index. For our proof-of-concept prototype, we use the 28,337 ungapped multiple alignments from BLOCKS database (version 14.1) as seed patterns, and run two iterations to generate 18,670 motif templates and 34 zone templates.

To organize an inverted index of motif and zone templates to UniProtKB (release 6.5) sequences, we use the open source MySQL relational database. The motif and zone templates are kept in two separate lists, where they are sorted by protein coverage. Each template has an associated critical detection score as defined by the last sequence database scan.

#### 3.5.2 Query Processing

Query processing operates in three phases: pattern matching, database search and results ranking.

In the pattern matching phase, we scan all patterns indexed by BioSpike over the query sequence. This phase generates a list of detected patterns, which are sorted ascendingly according to UniProtKB coverage sizes.

There are many variations of database search one can do given a set of detected
patterns. For example, the keywords-directed search as described in the Results Section limits the number of patterns in the database query. Since the patterns are sorted by ascending database coverage, as more “keywords” are included into the database query search, more remote homologues will be included in the result set. For our online search engine, we include all patterns in the search, allowing the user to select a result set size cutoff and bringing all detections that are in PDB to the top of the list.

For results ranking, we use a simple ranking function which compares the query sequence and result sequence in two perspectives: pattern detection score differences and pattern order differences.

Define the list of detected patterns in the query as an ordered list

$$\mathcal{A} = [(p_0, s_0), \ldots, (p_n, s_n)]$$

and the list of detected patterns on a result sequence as an ordered list

$$\mathcal{B} = [(\hat{p}_0, \hat{s}_0), \ldots, (\hat{p}_m, \hat{s}_m)]$$

For each list, we retain a single entry for each unique pattern $p_i$ (or $\hat{p}_j$) with the maximal score. We denote these reduced lists as $\mathcal{A}'$ and $\mathcal{B}'$.

To capture pattern detection score differences, we define $\mathcal{I} = (\mathcal{A}' \cap \mathcal{B}')_p$ as the intersection between the lists induced by $p_i = \hat{p}_j$. Finally, we define $\sigma$ and $\tau$ as ordered lists of patterns in $\mathcal{A}'$ and $\mathcal{B}'$, respectively, padding the shorter of the two with the missing patterns.

Using our notations, the ranking function is defined as,

$$\text{rank}(\mathcal{B}') = \frac{1}{|\mathcal{I}|} \sum_{p_i = \hat{p}_j} |s_i - \hat{s}_j| + C'_n(\sigma, \tau),$$

where $C'_n(\sigma, \tau)$ is the normalized Spearman footrule distance between the permutations $\sigma$ and $\tau$. Intuitively, the first term in the ranking function accounts for the difference in pattern detection scores, whereas the second term accounts for the difference in pattern arrangements in the primary sequences.
3.6 Results

This Results Section contains two parts. In the first we demonstrate that BioSpike can generate homology sets containing most of the set created by PSI-BLAST at a fraction of the computational cost. In the second we describe the results of two specific searches demonstrating that BioSpike can identify useful remote homologies that PSI-BLAST does not detect.

3.6.1 Comparison with PSI-BLAST

With a small set of patterns clustered and refined from the BLOCKS database (18,134 derived BioSpike patterns), we set the minimum significance of pattern detections to 100 (see Section 3.2) in order to cover 55% of the UniProtKB dataset. We extract 128 random sequences with at least 250 residues from this subset. Our benchmark comparison method is PSI-BLAST (version 2.2.13) with 6 iterations and all other default parameters. Using a single CPU machine running at 2.8 GHz, PSI-BLAST processes all 128 sequences in 17 hours. With the same system configuration, BioSpike processes all 128 sequences in 10 minutes. This is approximately a 100 fold improvement in speed.

To evaluate our search results, we compare the intersection sizes between the PSI-BLAST returned sequences ($P$) and the sequences returned from BioSpike ($B$). The retrieval rate is defined as $r = \frac{|P \cap B|}{|B|}$ and is measured at different levels of cut off for $B$. We note that this measure assumes the PSI-BLAST result set to be the gold standard, and does not account for homologous sequences returned by BioSpike but not by PSI-BLAST.

Figure 3.6 shows the retrieval rate at different levels of $|B|$ cut off. Table 3.1 shows the results of keywords-directed search. Keywords-directed search is when only a subset of the patterns found on the query sequence is used to search for homologous proteins in the database. On average, we can retrieve 72% of PSI-BLAST returned sequences with an approximately 100 fold improvement in speed.
Table 3.1: **BioSpike keywords-directed search statistics.** Patterns are ranked by lowest coverage first, and $k$ is the number of pattern “keywords” to incorporate in a protein database search.

<table>
<thead>
<tr>
<th>$k$</th>
<th>search time (s)</th>
<th>mean $r$</th>
<th>stdev $r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.5</td>
<td>0.4745</td>
<td>0.3527</td>
</tr>
<tr>
<td>2</td>
<td>3.6</td>
<td>0.6229</td>
<td>0.3360</td>
</tr>
<tr>
<td>3</td>
<td>3.7</td>
<td>0.6707</td>
<td>0.3390</td>
</tr>
<tr>
<td>4</td>
<td>3.8</td>
<td>0.6931</td>
<td>0.3462</td>
</tr>
<tr>
<td>5</td>
<td>3.8</td>
<td>0.6980</td>
<td>0.3475</td>
</tr>
<tr>
<td>6</td>
<td>3.9</td>
<td>0.7027</td>
<td>0.3472</td>
</tr>
<tr>
<td>all</td>
<td>5.0</td>
<td>0.7165</td>
<td>0.3479</td>
</tr>
</tbody>
</table>

Figure 3.6: The intersection size between PSI-BLAST returned sequences ($P$) and BioSpike returned sequences ($B$).
3.6.2 Results from specific searches

In this section we give two examples of specific searches.

Identifying the Legume lectin domains in a putative receptor-type protein kinase LRK1

We analyze the putative receptor-type protein kinase LRK1 extracted from the rice genome of *Oryza sativa*, accession number Q8H4I6. This protein has 636 amino acids and contains Legume lectin domains.

A PSI-BLAST search with 6 iterations and all other default options took 10.5 minutes and did not find any sequences with PDB annotations. With BioSpike, the sequence search took 8 seconds and found 9 PDB annotated sequences within the top 400 result sequences. Legume lectin alpha and/or beta domains are found in 8 of the 9 PDB annotated sequences.

The BioSpike template that was detected on these PDB annotated sequences is a cluster of 3 BLOCK patterns: Legume lectin alpha domain 1 (IPB000985F), Legume lectin alpha domain 2 (IPB000985G), and Legume lectin beta domain 1 (IPB001220G). BLOCK IPB000985F is the anchor pattern on the template; BLOCKs IPB000985G and IPB001220G are patterns in position 36 of the tem-
Figure 3.8: **Legume lectin domains in several BioSpike result sequences.** The protein sequences are colored in blue, and the detected Legume lectin domains are highlighted in red. Related PDB chains are colored in green.
plate. To illustrate the versatility of the BioSpike template, we show a multiple alignment of the PDB sequence regions indexed by this template in Figure 3.7. Clearly, the BioSpike template captured the various gaps between the two distinctive Legume lectin domains. Figure 3.8 also illustrates the 3-dimensional structures for 4 of the PDB annotated regions generated by QuickPDB.

**Identifying the Enoyl-CoA hydratase site of a putative enzyme**

We analyze the putative enzyme extracted from the bacterial genome of *Shigella flexneri*, accession number Q83QQ0. This protein has 714 amino acids and contains a domain with an Enoyl-CoA hydratase site.

A PSI-BLAST search with 6 iterations and all other default settings took 11.5 minutes and did not find any sequences with PDB annotations. With BioSpike, the sequence search took 5 seconds and found a PDB annotated sequence with the Enoyl-CoA hydratase site. Using the PDB sequence returned, one can easily infer the structure of the Enoyl-CoA hydratase site in the putative enzyme.

### 3.7 Conclusion

We have presented a novel method for searching protein homologues using amino acid patterns as indexes. Our method is approximately 100 times faster than PSI-BLAST whilst maintaining a similar level of accuracy. Moreover, the ability of BioSpike to identify remote homologues is superior to that of PSI-BLAST. The main reason that BioSpike is so fast is that we rely heavily on pre-computation. By pre-computing and storing on disk the locations of pattern detections, we remove the need to perform similar computations repeatedly. Pre-computation also creates a dramatic reduction in the time it takes the system to respond to a new query. This reduction in query processing time can, we believe, make the system much more attractive as an interactive exploration tool for the research biologist.

We are considering many directions for continuing this work. This list includes enlarging, refining, and curating the set of motifs; creating data structures to
speed up the search for motifs in a new query sequence; using motifs as features for classifying proteins; and identifying which motifs have consistent 3D structure and can thus be seen as “building blocks” of proteins.

Acknowledgments

The text of this chapter, in part, is a reprint of the material as it appears in the University of California, San Diego, CSE Technical Reports CS2006-0858, 2006 [44]. The dissertation author was the primary author, and the co-author listed in this publication supervised the research.
4

Motif Finding using Random Projections

Protein motifs are well conserved sequence patterns that often code for essential structural or functional components of proteins. Since the availability of high throughput sequencing tools, the number of known protein sequences have been growing at an unprecedented rate while structural and functional annotations remain extremely sparse. In Chapter 3, we discussed one of the methods to bridge this knowledge gap, namely by transferring annotations through sequence homology using a motif-based index. The idea is that protein sequences sharing similar sequence motifs should expect to share similar structural or functional traits in the domain or sub-domain level. Our discussions assume that we start with a library of curated multiple sequence alignment and we provided a framework to build a statistically robust motif index.

In this chapter, we remove this assumption by bootstrapping our motif indexing framework with multiple sequence alignments automatically generated from a large sequence database. Since we desire small indices with large database coverage, our focus is on sequence patterns that are prevalent across many protein sequences. This focus is very different from the focus of most well known motif finding algorithms. In particular, most motif finding algorithms are evaluated in terms of how well they can find artificially planted subtle motifs with weak statistical supports. Hence, the emphasis of these algorithms is in the design of sensitive objective functions to direct their search algorithms. This is discussed in detail in a review by Li and Tompa [58].
Our goal focuses on discovering motifs that are prevalent across protein space, and hence the objective function plays a lesser role in our design considerations. Instead of focusing on a sensitive search function, we concentrate on designing an algorithm to efficiently group similar sequence signals into buckets for alignment analysis. In this regard, the motif finding methods presented by Buhler et al. is most fitting [15, 17, 16]. In a series of paper, Buhler et al. demonstrated the effectiveness of using random projections and the theory of locality sensitive hashing [34] for motif discovery.

This chapter extends this line of work by incorporating statistical analysis of random projections to discover novel motif families. Our method for motif family discovery is efficient, makes very little assumptions and is easily automated. We present a systematic way to index any sequence database using motif families. Motif families form rough signals that can reduce the size of the motif-based database indexes, giving searches an additional advantage in terms of speed without compromising the sensitivity of motif directed homology search. We begin with a short review on the theory of random projections and motif finders that use it.

4.1 Background and Related Work

Random projection is a powerful dimensionality reduction method. High dimensional data is embedded in a lower dimensional subspace and the distribution of these embedding is analyzed. In essence, given a matrix $X_{m \times n}$ consisting $n$ data points in $\mathbb{R}^m$, the dimensionality reduced data set can be obtained by projecting $X$ onto a lower dimensionality subspace using a transformation matrix, $X'_{k \times n} = T_{k \times m} \cdot X_{m \times n}$, where $k$ is the desired dimensionality.

The Johnson-Lindenstrauss lemma [48] states that when the set of examples in $X_{m \times n}$ is mapped onto a $O(\log n / \epsilon^2)$ dimensional subspace with $0 < \epsilon < 1$, the probability of distorting Euclidean interpoint distances by more than a factor of $(1 \pm \epsilon)$ is $O(1/n^2)$. Specifically for any mapping function $f : \mathbb{R}^m \rightarrow \mathbb{R}^k$ where
\[ k = \mathcal{O}(\log n/\epsilon^2) \], any pair of columns \( u \) and \( v \) in \( X_{m \times n} \) will observe,

\[
(1 - \epsilon)\|u - v\|_2^2 \leq \|f(u) - f(v)\|_2^2 \leq (1 + \epsilon)\|u - v\|_2^2
\]

with high probability. The full probabilistic proof is provided by Dasgupta and Gupta [20]. The lemma motivates efficient methods for analyzing any data in high dimensional space by circumventing the curse of dimensionality. In Figure 4.1, we show two different random projections of high dimensional data onto a simple one dimensional line. It is obvious that different random projections will produce a slightly different distribution of the projections, but in general both random projections preserve certain aspects of the distribution of high dimensional data.

![Figure 4.1: Two randomly chosen projections of the same high dimensional data points. The blue boxes indicate that the distribution of data points on the one dimensional line is dependent upon the projection.](image)

Buhler’s motif finding work has focused on the analysis of genomic sequences to solve the planted motif problem formulated by Pevzner and Sze [71]. In a \((l, d)\)-planted motif problem, a consensus motif sequence model \( M \) of length \( l \) is fixed and planted in every \( t \) sequence fragments each of length \( n \). Prior to implanting the consensus sequence into the length \( n \) fragment, exactly \( d \) positions are corrupted. The search algorithm must recover the location of these implants and the hardness of the planted motif problem for traditional motif finders is discussed in detail in Buhler’s presentation of his PROJECTION algorithm [17].
The PROJECTION algorithm essentially iterates between two phases: *search* and *refinement*. First all overlapping $l$-mers are extracted from the sequence set for analysis. In the search phase, a mapping function $f$ defines $k$ random unique positions ($i_1 \ldots i_k$) where $k < l$. The map $f$ projects the fragments from the $l$-mer space to a lower dimensional subspace for sequence comparison. The $k$ random positions define the exact locations where $l$-mers must match. A bucket is defined as a set of $l$-mers that have identical nucleotides in all $k$ positions. Buckets with high concentration of $l$-mers are interesting as they probably contain the motif of interest. These high concentration buckets move on to the refinement phase for further analysis. The refinement phase can be supplemented by any local search techniques common to many traditional motif finding algorithms. In essence, PROJECTION is an initialization algorithm that avoids the issue of determining a good seed candidate — a problem that usually determines the success of any local search algorithms. An example of such an analysis is illustrated in Figure 4.2.

![Random projection algorithm for genomic sequences.](image)

**Figure 4.2:** Random projection algorithm for genomic sequences. In each successive stage (top to bottom), we fix random positions in the 8-mer patterns. We show that the buckets become more specific in each iteration.

The alphabet for DNA sequences contains only four nucleotides and a system of match/mismatch is sufficient for modeling sequence similarity. However, this
model for sequence comparison is not appropriate for protein sequences with an alphabet of twenty residues. Substitution rates amongst the twenty amino acids are commonly expressed by the BLOSUM or PAM matrices [41, 21]. These matrices capture a richer model of residue substitution patterns common to representative protein family alignments. As expected, hydrophobic residues substitute more frequently with other hydrophobic residues but not as readily with the other hydrophilic counterparts. It is important to incorporate substitution patterns into the random projections-based motif finders to accurately estimate the Hamming distances between protein fragments.

Mathematical formulations for incorporating substitution matrices into the random projection-based motif finders were first defined by Buhler [16] and later concretized by Halperin et al. [36]. In this chapter, we provide a more direct method for embedding amino acids in Euclidean space based on any substitution matrices, and use the embedding to calculate sequence similarity based on any random projections. We also do not focus on the refinement of the motifs found, but use the random projection algorithm to discover prevalent protein patterns for indexing large sequence databases.

4.2 Our Contributions

We first discuss how to embed residues and protein fragments in Euclidean space. This is followed by a description of how we apply random projections to find recurring motifs in protein sequences. Finally, we discuss how the method can be used to automatically index large protein databases.

4.3 Sequence Embedding

Denote $\Sigma$ as the alphabet for protein sequences. Given any log-odds substitution matrices $S_{|\Sigma| \times |\Sigma|}$, such as BLOSUM62, we can convert it into a approximate distance matrix $D_{|\Sigma| \times |\Sigma|} = \max_{ij} S - S$, where $\max_{ij} S$ is the maximal element
in the log-odds matrix \( S \). The resulting matrix \( D \) is not a true distance matrix since the triangle inequality is not guaranteed. However this affine translation sufficiently approximates the “distance” between residues in terms of substitutability, resulting in a softer version of the match/mismatch rule.

Once we have defined the matrix \( D \), we apply classical multidimensional scaling (cMDS) to discover an embedding of the alphabets in Euclidean space. cMDS, a form of principle components analysis (PCA), is another dimensionality reduction method that discovers the greatest direction of variance in a matrix of interpoint distances. These directions are expressed as a set of eigenvector and eigenvalue pairs, where the eigenvalues rank the eigenvectors by importance. Ideally, only two to three eigenvectors (i.e. dimensions) are sufficient to approximately reproduce the interpoint distances in the original (dis)-similarity matrix. In Figure 4.3 we show an embedding of the amino acids based on cMDS analysis of an affine translated BLOSUM62 matrix.

Assume that cMDS gives us \( m \) eigenvalue and eigenvector pairs, where each eigenvector \( \vec{v} \in \mathbb{R}^{\mid \Sigma \mid} \). Denote \( \vec{v}_i \) as the \( i \)th eigenvector and each eigenvector has dimensions \((\mid \Sigma \mid \times 1)\). By choosing the first \( n \) strongest eigenvectors, we can express each amino acid as a point in \( n \)-space. Precisely, each row in the matrix \( C_{\mid \Sigma \mid \times n} = (\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n) \) corresponds to a coordinate of the amino acid embedding. This embedding is described as the mapping \( \phi : \Sigma \rightarrow \mathbb{R}^n \). For a fragment \( x \) of length \( l \), the fragment embedding is \( \hat{\phi}(x) = \langle \phi(x_1), \phi(x_2), \ldots, \phi(x_l) \rangle \). That is, the fragment embedding is defined by the map \( \hat{\phi} : \Sigma^l \rightarrow \mathbb{R}^{ln} \). Similar fragments \( x \) and \( y \) will have small angles between their corresponding embeddings \( \hat{\phi}(x) \) and \( \hat{\phi}(y) \).

Instead of using all \( l \) residues in a fragment, we select a random subset of positions to embed in Euclidean space. In particular, we randomly pick \( k \) of the \( l \) position and denote these positions as \((i_1, \ldots, i_k)\). The corresponding symbols we extract from \( x \) is \( x' = x_{i_1} \ldots x_{i_k} \). The embedding of fragment \( x \) under this random projection is defined as \( \phi_{(i_1, \ldots, i_k)} = \langle \phi(x_{i_1}), \phi(x_{i_2}), \ldots, \phi(x_{i_k}) \rangle \). In summary, the random projection embedding is defined by the map \( \hat{\phi}_{(i_1, \ldots, i_k)} : \Sigma^l \rightarrow \Sigma^k \rightarrow \mathbb{R}^{kn} \). The next section discusses the linear projection analysis we perform on these
Figure 4.3: **An embedding of amino acids in 3-space.** The hydrophobic residues (mostly on the left) are clearly separated from the hydrophilic residues (mostly on the right). The symbol X is standard for denoting unknown residues in sequence data. It is clearly positioned in the center for all residues.

4.4 Random Linear Projections

One way to compare the fragment embeddings is to project all of them onto a number line. Specifically, we initialize a random vector $\vec{u} \in \mathbb{R}^{kn}$ by concatenating $l$ unit vectors in $\mathbb{R}^{n}$, each randomly sampled over a unit ball using the normal
distribution $\mathcal{N}(\bar{0}, I)$. The random projection of a fragment $x$ is defined as

$$h_{\bar{u}}(x) = \bar{u} \cdot \phi_{(i_1 \ldots i_k)}(x),$$

where similar fragments $x$ and $y$ observe the property that $|h_{\bar{u}}(x) - h_{\bar{u}}(y)| < \epsilon$ for some small $\epsilon > 0$. The parameters $\bar{u}$ and $(i_1 \ldots i_k)$ is fixed over a single analysis of all $l$-mers in the sequence database. These parameters define a unique set of recurring sequence patterns.

Given a random sample of sequence fragments, we compute their projection values to determine the sample mean ($\mu$) and variance ($\sigma^2$). These sufficient statistics forms the basis of our null model. We expect that projections based on $\bar{u}$ and $(i_1 \ldots i_k)$ will be distributed according to the distribution $\mathcal{N}(\mu, \sigma)$ with minor distortions. In reality, however, certain fragments will be over represented because of evolutionary pressure. This observation motivates our motif search method.

We construct empirical and null histograms where each bin holds 0.05% of random projection values based on the $\mathcal{N}(\mu, \sigma)$ distribution. For the null model, we simulate random sequence fragments based on the random sequence model where each position is independently generated from the amino acid background distribution. Each random sequence fragment is mapped to a projection value based on $\bar{u}$ and $(i_1 \ldots i_k)$ and the values are collected for the null histogram. The null histogram should be relatively flat as each histogram bin is designed to hold a fixed amount of projection values, but fluctuations are inevitable due to sampling noise. For the empirical model, we collect projection values for all fragments in a database based on the same parameters $\bar{u}$ and $(i_1 \ldots i_k)$. The ratio between the empirical count and the null count represents the level of over-representation of a particular fragment configuration. Figure 4.4 illustrates a typical analysis. It is important to note that the significance of our findings should be measured in terms of the fluctuation variance for the null and empirical models. The null model has minor fluctuations, whereas the empirical model observe larger fluctuations suggesting a more disproportionate partitioning of the sequence space.

To harvest motifs, we first sort the histogram bins and inspect the top $m$ bins
with greatest empirical-to-null count ratios. Each bin is defined by a minimum and maximum projection value and represents a cluster of sequence fragments with similar composition. Using the projection value bounds, we can aggregate motif fragments in another full database scan. Neighboring bins with positive non-unity ratios can also be aggregated and this process allows us to create rough sequence motif models, forming the basis of a sparse motif index.

4.5 Motif Indexing Framework

Using the approach above, the indexing of large sequence database with statistically meaningful motifs becomes straightforward. Given any large sequence database, we iteratively execute the motif finding algorithm with different parameters $\vec{u}$ and $(i_1 \ldots i_k)$. The parameter $k$, the number of fixed location in a fragment for projection, and the parameter $l$, the length of the motif, can be tuned for flexible motif lengths with different number of configuration constraints. At each execution of the motif finding algorithm, we collect fragments only from the top $m$ histogram bins representing $m$ new motif models in the index. The location of these fragments are marked in the database and will not be considered in subsequent motif finding runs. In this way, the sequence fragments under consideration will continually decrease in number until no more fragments can be grouped.

Instead of collecting all $m$ motif models in every execution, one might consider selecting a subset of motifs with evidence of statistical significance. In each execution of motif finding, the positions $(i_1 \ldots i_k)$ are called the fixed locations and all other positions are called the free locations. We compare the average Kullback-Leibler (KL) divergence of the fixed and free locations with the amino acid background distribution. When the KL divergence is high for both the fixed and free locations, one can be quite certain that the model encodes for an evolutionarily conserved sequence configuration. This situation is most ideal when $k$ is relatively small compared to $l$. Figure 4.5 shows an example of one such analysis.
Figure 4.4: **Histogram comparisons between the empirical and null models.** The empirical model is based on random projection values for length 30 fragments in a database, where 20 of the 30 positions are fixed for sampling and each residue is mapped to a point in $\mathbb{R}^5$. 
Figure 4.5: **KL analysis of fixed and free positions along a motif model.** Vertical red dashed lines signify the fixed positions (top) and the motif model is summarized in a frequency matrix (bottom).
5

Gene Finding in Microbial Genome Fragments

Since the introduction of 16S rRNA gene sequencing [69, 68], the study of microbes have slowly shifted away from standard laboratory culture to environmental sampling. Instead of investigating microbes in isolation, metagenomics initiatives study bacteria species in their natural habitats. The approach combined with recent advances in massively parallel sequencing technologies revolutionized the field of microbiology and opened up the possibility of research in the estimated 99% of understudied microbes [43]. Whole communities of microbial organisms can now be analyzed as a snapshot, enabling the research of phylogenetic relationships and metabolic pathways of microbes in specific environments.

Currently the most popular sequencing approach to metagenomics analysis is the whole genome shotgun sequencing method. This sequencing method has been applied to various metagenomics projects ranging from simple community models, such as the acid mine biofilm project containing 5 species [88], to highly complex community models such as the Sargasso Sea environmental sampling project containing an estimated 1,800 species [90]. While whole genome shotgun analyses is efficient in extracting DNA fragments, reconstructing full genomes from these fragments typically requires a large amount of sequencing. In the acid mine biofilm project, the number of species in the community is small and sequencing 65 Mbps is sufficient in producing many sequence fragment overlaps for the genome recon-
struction of two dominant species. In the Sargasso Sea project, 50% of the 1.6 Gbps sequenced are single reads averaging 818 bps in length and genome reconstruction are only possible for three of the estimated 1,800 species. As the field of metagenomics develops, it is plausible that the community models under investigation will become increasingly complex and genome reconstruction is probably not a practical approach for sequence fragment pre-processing.

However, genome reconstruction is an essential preprocessing step for most prokaryotic gene finders that are currently available. Prokaryotic gene finders such as GLIMMER [78, 22] and EasyGene [54, 66] are based on hidden Markov models (HMMs) and their application is specific to individual microbial species. Building these HMMs requires ample amount of representative sequences specific to the organism and this requires an impractical amount of sequencing when the metagenomics environment contains thousands of species. This limits the application of HMM gene finders on metagenomics sequencing projects.

Instead of training individual models for each microbial species, we propose a predictor that integrates features commonly found upstream and downstream on curated prokaryotic genomes. Most of the patterns we mine are correlated with microbes of different GC bins so as to generalize a system of weak classifiers irrespective of species origin. The weak classifiers are adaptively combined using boosting to form a strong final classifier that can distinguish codons from correct and incorrect reading frames.

The rest of the chapter is organized as follows. First, we provide a brief overview of conventional gene finding approaches and discuss the relationship of our work with recent research on metagenomics gene finding. This is followed by an overview of the machine learning tools we use to learn codon classifiers. We next setup our problem and introduce the list of weak learners we combine to form our codon classifiers. We then provide experimental results on gene finding, comparing our approach to MetaGene [67] as well as illustrating the performance on another simulated metagenomics data set.
5.1 Gene Finding for Metagenomics

Conventional gene finding approaches on prokaryotic genomes can be divided into two categories: intrinsic and extrinsic. Intrinsic methods such as GLIMMER [78, 22] and GeneMark [11] use fundamental sequence characteristics to discriminate coding and non-coding open reading frames (ORFs). The genetic code is a redundant code with 61 codons mapping 20 amino acids, and the genome GC content naturally imposes codon usage biases. To tackle the large range of GC content in microbial genomes (25-75%), most current intrinsic methods need to model each bacteria genome separately.

Instead of figuring out the fundamental compositional differences in coding and non-coding regions, extrinsic methods such as ORPHEUS [32] and CRITICA [5] perform similarity matching of the query sequence with currently available bacterial annotations to discover potential coding regions. The key assumption is that evolution of genes are directed by gene duplication, rearrangement and mutation of existing genes and hence similarity between coding regions are expected. Finally there are hybrid approaches, such as the framework provided by McHardy et al. [63], that combines both strategies.

Recently two methods have been implemented to deal with metagenomics sequences. Krause et al. devised an approach that generalizes extrinsic methods to tackle sequencing errors in metagenomics sequences [51]. In summary, each position of a query fragment is associated with 7 scores that corresponds to different frame shifts. An optimal path that models the most likely sequence of frame shifts is computed by dynamic programming. This in turn allows the algorithm to infer the coding region(s) on the query sequence.

MetaGene takes an approach that is more reminiscent to intrinsic gene finding techniques [67]. All possible candidate ORFs are scored on a query fragment using a suite of scoring functions that measure various sequence properties indicative of protein codingness. These include GC content and amino acid frequencies correlations [11], dicodon frequencies, and ORF length. As a final step, MetaGene
exhaustively searches for a subset of these candidate ORF with the highest likelihood. Both approaches above were evaluated on synthetic metagenomics data set and have been shown to work well.

### 5.2 AdaBoost

AdaBoost is a general discriminative learning algorithm invented by Freund and Schapire [31]. The basic idea of AdaBoost is to repeatedly apply a simple learning algorithm, called the *weak learner*, to different weightings of the same training set. In its simplest form, AdaBoost is intended for binary prediction problems where the training set consists of pairs \((x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\), where \(x_i\) corresponds to the features of an example, and \(y_i \in \{-1, +1\}\) is the binary label to be predicted. A *weighting* of the training examples is an assignment of non-negative real value \(w_i\) to each example \((x_i, y_i)\).

On iteration \(t\) of the boosting process, the weak learner is applied to the training set with a set of weights \(w_1^t, \ldots, w_m^t\) and produces a prediction rule \(h_t\) that maps \(x\) to \{0, 1\}. The requirement on the weak learner is for \(h_t(x)\) to have a small but significant correlation with the example labels \(y\) when measured using the current weighting of the examples. After the rule \(h_t\) is generated, the example weights are changed so that the weak predictions \(h_t(x)\) and the labels \(y\) are de-correlated. The weak learner is then called with the new weights over the training examples, and the process repeats. Finally, all of the weak prediction rules are combined into a single *strong* rule using a weighted majority vote. One can prove that if the rules generated in the iterations are all slightly correlated with the label, then the strong rule will have a very high correlation with the label — in other words, it will predict the label very accurately.

The whole process can be seen as a variational method in which an approximation \(F(x)\) is repeatedly changed by adding to it small corrections given by the weak prediction functions. In Figure 5.1, we describe AdaBoost in these terms. We shall refer to \(F(x)\) as the *prediction score* in the rest of the paper. The strong
Figure 5.1: The AdaBoost algorithm.

The prediction rule learned by AdaBoost is \( \text{sign}(F(x)) \).

### 5.3 Alternating Decision Trees

AdaBoost is often used with a decision tree learning algorithms as the base learning algorithm. We use AdaBoost both to learn the decision rules constituting the tree and to combine these rules through a weighted majority vote. The form of the generated decision rules is called an alternating decision tree (ADT) [30].

We explain the structure of ADTs using one of the trees obtained after running 10 iterations of boosting in Figure 5.2. The problem domain is the codon reading frame classification and the goal is to predict whether a codon is in the correct reading frame of a gene. It is based on indicators such as coding potentials, coding length, and existence of protein motifs. The tree consists of alternating levels of ovals (prediction nodes) and rectangles (splitter nodes). The numbers within the ovals define contributions to the prediction score. In this example, positive contributions are evidence of a in-frame codon, negative contributions are evidence of an out-of-frame codon. To evaluate the prediction for a particular codon we start at the top oval \((-1.384, \text{ i.e. assume it is out-of-frame})\) and follow the arrows down. We follow all of the dotted arrows that emanate from prediction nodes, but we follow only one of the solid-line arrows emanating from a splitter node, corresponding to the answer (yes or no) to the condition stated in rectangle. We
Figure 5.2: An ADT for the problem of classifying in-frame codons from out-of-frame codons after 10 iterations of boosting.
sum the values in all the prediction nodes that we reach. This sum represents the prediction score $F(x)$ above, and its sign is the prediction.

For example, suppose we had a codon for which $\text{CodonUsage}=1.3, \text{Frame0}=0.4, \text{Frame2}=-0.5, \text{Frame}4=-0.8, \text{NumMotifs}=5, \text{Length}=120$. In this case, the prediction nodes that we reach in the tree are -1.384, +0.864, +0.805, +0.313, +0.561, +0.477, +0.532, +0.627, +0.293, +0.667, and summing gives a score of 3.755, i.e., concrete evidence that the codon is in the correct reading frame of a coding region.

The ADT in the figure was generated by AdaBoost from training data. In terms of AdaBoost, each prediction node represents a weak prediction rule, and at every boosting iteration, a new splitter node together with its two prediction nodes is introduced. The splitter node can be attached to any previous prediction node, not only leaf nodes. Each prediction node is associated with a weight $\alpha$ that contributes to the prediction score of every example reaching it. The weak hypothesis $h(x)$ is 1 for every example reaching the prediction node and 0 for all others. The number in front of the conditions in the splitter nodes of Figure 5.2 indicates the iteration number on which the node was added. In general, lower iteration numbers indicate that the decision rule is more important. We use this heuristic to analyze the ADTs and identify the most important factors in finding coding regions on metagenomics sequences.

### 5.4 Problem Setup

Our problem focuses on identifying codons that are in the correct reading frame by formulating a binary classification problem. The positive class are codons that are part of coding regions and in the correct reading frames. The negative class consists of all other codons. To reduce the size of the example space, we define marker codons to start a region of interest and stop codons to terminate it. Using all possible ORFs on a metagenomics fragment is a special case of this, in which start codons are used as marker codons. We use common start codons as marker codons in our experiments, but essentially one can use any codons to make it
flexible for predicting regions in erroneous fragments. From here-on, we refer these regions of interest as ORFs.

Given a metagenomics fragment, we extract all possible candidate ORFs by considering the common start and stop codons on both positive and negative strands. The start codons we consider are ATG, TTG, and GTG; and the stop codons we consider are TAA, TAG, and TGA. The classification problem assigns positive labels to the 3 types of codons that are part of a coding region. It does not provide information about the exact start of a gene. Regions lacking stop codons (or partial ORFs) are also considered if they are at least 60 bps in length.

In addition to using start codons to restrict our example space, we perform further reduction by considering subsets of candidate ORFs. A metagenomics fragment may contain multiple candidate ORFs expressing part of the same coding region. We define two candidate ORFs to be in the same set if they are in the same frame and they both share (or lack) a stop codon. We group ORFs of the same set and only pick the longest \( k \) for classification. In this way, not all candidate markers on a segment is considered and more importantly, our estimation of sequence properties are averaged over longer segments. Figure 5.3 illustrates how to formulate positive and negative examples when start codons are used as markers codons.

5.5 Weak Learners

In this section, we present the suite of weak learners that we encode for boosting. Each weak learner captures a specific characteristic that may help a final classifier decide if a candidate marker is in the correct reading frame of a gene. We design the suite such that features are mostly orthogonal to each other as it is preferable that weak learners are not strongly correlated. Figure 5.4 illustrates how the weak learner processes each metagenomics fragment. We only use curated protein coding regions for building weak classifiers, i.e. proteins not described as hypothetical, possible, probable, predicted, putative, and unknown.
Figure 5.3: **Classification problem for metagenomics gene finding.** The position with the first 0 is an annotated gene start. Circled positions are positive examples and rhombus-marked positions are negative examples.

Figure 5.4: **Describing features on metagenomics fragments.** \((x_i, y_i, w_i)\) describes example \(x_i\) with label \(y_i\) and initialized with weight \(w_i\). The weak learner, \(h\), scores the example based on a particular feature. It will be combined with other weak learners in boosting to provide a final classification.
5.5.1 Basic Fragment Properties

Three basic features we encode as weak learners are the GC content, the type of marker used, and if available, the type of stop codon used. GC content is useful in indicating the possible bin of species the fragment came from. A stronger model which we describe below correlates GC content with codon usage tables. It exploits the fact that species within certain GC bins have similar codon usage patterns. The GC content bins we explore are 20–39%, 40-49%, 50-59%, and 60-79%.

5.5.2 Codon Usage Patterns

GC content of a genome often correlates with the genome’s codon usage patterns. For example, a genome with high GC content usually observes higher frequencies of GCG and GCC codons in coding regions. The design of heuristically generated codon usage tables were explored by Besemer et al. [11]. By using a small sample of DNA fragments from different genomes, they were able to correlate GC content with amino acid frequencies using linear regression models.

We incorporate this knowledge into our boosting setup by constructing general codon usage tables for each GC bin. For each GC bin, we extract annotated protein coding regions and collect the codon frequencies as our foreground model, \( \vec{f} = (f_{AAA}, \ldots, f_{TTT}) \). As background model, we collect the codon frequencies used in all random ORFs, \( \vec{q} = (q_{AAA}, \ldots, q_{TTT}) \). Random ORFs are defined by any consecutive codon sequence with a valid start and stop codon. Finally, we construct the codon usage table as a set of log-odds scores, \( \vec{l} = (\log f_{AAA} - \log q_{AAA}, \ldots, \log f_{TTT} - \log q_{TTT}) \). Scoring examples involve choosing the correct codon usage table, and this is estimated by the fragment’s GC content. Currently available metagenomics fragments (except for those from pyrosequencing) are usually 700-1,000 bps in length, and this amount of data should provide us with a statistically good estimate of the fragment’s origin.

To score a marker codon, we extend a region downstream to form a full or partial ORF. For a candidate region with \( n \) codons, \( x_1y_1z_1 \ldots x_ny_nz_n \), we com-
pute the coding potential defined as the average of codon usage scores, \( L_1 = \frac{1}{n} \sum_{i}^{n} (\log f_{x_i,y_i,z_i} - \log q_{x_i,y_i,z_i}). \) We also compare the score for the 5 other surrounding frames, \((L_2, L_3, \ldots, L_6)\). As the output of this weak learner, we use \( L_1 - \max_{i \neq 1} L_i \) to highlight the relative codingness of the region extended from the marker codon with respect to its surrounding frames. True ORFs are expected to observe higher relative scores.

### 5.5.3 Exploring Structures Within Codon Usage Patterns

We extend our codon usage analysis by exploring subfamilies of codon usage patterns in each GC bin. This is of particular interest as it is well known that certain genomes, such as the *Escherichia coli* genome, contain different classes of proteins that are coded by different codon usage tables. For example, genes found in the *E. coli* genome can be divided into three classes [64]. Class I and Class II genes have similar codon usage patterns, whereas Class III genes, presumably horizontally transferred, use a very different set of codons. Instead of generalizing an entire GC range of species with one overall codon usage table, we cluster ORFs for genomes in a specified GC range to discover more interesting codon usage patterns for *classes* of genes. Figure 5.5 illustrates a hypothetical embedding of the proteins in *E. coli* in 2-space.

To ensure fairness in performance evaluation, the codon usage patterns we build are based only on genomes in the training set. Genomes are first partitioned into the different GC bins. We then extract coding regions for curated proteins and represent each of them as a sparse 64-dimensional probability vector. These vectors describe the frequency of codons found along ORFs and we cluster these vectors to reveal interesting codon usage patterns.

In our experiments, we use the \( k \)-means clustering algorithm with \( k \) tuned by cross-validation. The algorithm is initialized with \( k \) random codon usage tables. Class membership for points in the 64-dimensional space is determined greedily by the choosing the center that provides the maximum log-likelihood. After class
labels are assigned to all points, the cluster centers are re-estimated. The algorithm iterates over these two steps until the class membership amongst the set of points stabilizes.

At algorithm termination, we obtain $k$ codon usage tables for each GC bin. As with above, the GC content of a metagenomics fragment determine which set of $k$ codon usage tables to use for scoring. To score a marker codon, we extend a region downstream to form a full or partial ORF. For the candidate region of $n$ codons, $x_1 y_1 z_1 \ldots x_n y_n z_n$, we compute $k$ coding potential scores as an average of codon usage scores. The output of this weak learner is

$$\max \left\{ \frac{1}{n} \left( \sum_i \log f^k_{x_1 y_1 z_1} - \log q_{x_1 y_1 z_1} \right), \ldots, \frac{1}{n} \left( \sum_i \log f^k_{x_n y_n z_n} - \log q_{x_n y_n z_n} \right) \right\}.$$  

5.5.4 Generalizing Codon Patterns over Multiple Frames

So far our discussion of codon patterns have been limited to codons that are in the correct reading frame. Our final extension involves modeling codon distributions for the 5 other incorrect reading frames. The construction of these codon
models follows similar methodologies described previously in Section 5.5.3. Given an annotated protein coding region, we look at the codon frequencies for the 6 possible frame reads. These codon frequencies are represented as probability vectors to be clustered in a 64-dimensional space using the $k$-means algorithm. The end result of our construction is $6k$ codon frequency tables for each GC bin.

As with the previous weak learners, we choose the correct $6k$ codon frequency tables by estimating the GC content of a metagenomics fragment. To score a marker codon, we extend a region downstream to form a full or partial ORF. For the candidate region $x_1y_1z_1 \ldots x_ny_nz_n$, we compute the maximum log-likelihood score for each of the 6 frame models,

$$L_j = \max \left\{ \frac{1}{n} \left( \sum_i \log f_{x_iy_iz_i}^1 - \log q_{x_iy_iz_i} \right), \ldots, \frac{1}{n} \left( \sum_i \log f_{x_iy_iz_i}^k - \log q_{x_iy_iz_i} \right) \right\},$$

for $j = 1, \ldots, 6$. This gives us 6 scores ($L_1, \ldots, L_6$), each representing a particular frame model. As a final step, we compare how each of the frame model explains the origin of the marker codon by computing the set of relative scores ($L_1 - \max_{i \neq 1} L_i, \ldots, L_6 - \max_{i \neq 6} L_i$).

### 5.5.5 Modeling Protein Signatures by Context Trees

Our boosting framework incorporates variable length Markov models to discover common protein signatures. Currently, the most well known method that constructs context trees for prokaryotic genomes is GLIMMER [78]. GLIMMER builds interpolated Markov models (IMMs) that capture common patterns in nucleotide streams. The idea is to model nucleotides patterns of different lengths, utilizing longer patterns whenever there is sufficient data. Depending on the length of the DNA oligomer query, the context of the sequence model adapts to output a likelihood score based on the local sequence composition.

We build context trees on the level of amino acids to model common physicochemical patterns on real proteins. One such pattern that we capture is the alternating sequences of hydrophobic and hydrophilic residues. Distinct patterns of
this sort are prevalent in certain protein motifs, such as those composed of multiple alpha helices. Examples are surface helices and water transport helices. The former structure exposes one side of the helix to the external hydrophilic environment, while the other side of the helix interfaces the predominantly hydrophobic core. The latter structure must maintain a predominantly hydrophilic helical core for water transport. The functional and structural requirements of these motifs imposes a conservation constraint on the primary sequence, and hence the primary sequences should observe distinct hydrophobic and hydrophilic patterns.

As with our previous weak learners, we build foreground and background models to compare sequence likelihoods. In this case, we construct a model using the proteins translated from annotated gene segments and 5 other models using pseudo-proteins translated from surrounding incorrect reading frames. The former represents our empirical model that captures patterns in real proteins, while the latter set represents our background model that encode various types of noise around protein coding regions. Translated proteins and pseudo-proteins are converted to bit sequences to represent alternating hydrophobic/hydrophilic states. These bit sequences are the inputs to building context tree models, and also the inputs when using the models to evaluate sequence likelihood.

Denote the models output log likelihood scores, \( (L_1, \ldots, L_6) \). Our weak learner correspondingly outputs 6 scores evaluating the relative strength of the models given the observed sequence, \( (L_1 - \max_{i \neq 1} L_i, \ldots, L_6 - \max_{i \neq 6} L_i) \). For evaluation fairness, we build models only using metagenomics fragments in the training set.

### 5.5.6 Coding Length

Another important feature that can often help distinguish true ORFs from random ORFs is the length of the coding region. Recall that we only consider the longest regions in a ORF subset for classification and this means that length is possibly an important feature for scoring candidate regions. Longer regions with high codon usage scores provide better evidence of a valid coding region. We use
both the actual lengths measured in amino acids as well as log-likelihood scores based on the distribution of true and random ORF lengths.

The construction is similar to subsequent weak learners we design. The first step consists of listing all possible ORFs for genomes in the training set. We build a histogram for true ORF lengths as well as background histograms, categorized by different genome GC bins, of false ORFs lengths. For the background model, we also interpolate any undefined intermediate ORF lengths. The log ratio between the empirical and background ORF length distribution is our log odds scoring table for evaluating candidate regions on the basis of coding length. For a given metagenomics fragment, the fragment GC content directs the use of a particular background model for scoring.

5.5.7 Motif Finding on Fragments

Finally, we also incorporate protein motif indexes into our boosting framework. The use of this feature resembles techniques used in extrinsic gene finding approaches. Extrinsic features such as these provide an extra boost to positive prediction scores made using intrinsic patterns. Instead of using BLAST to search for gene coding regions in annotated microbial genomes, we perform motif lookups using the libraries we built in Chapter 3 and Chapter 4. Since the evolution of genes are directed by the process of gene duplication, rearrangement and mutation, many of the key motif elements must be conserved and shared across microbial species. If the amino acid translation of a candidate ORF contains commonly found motif signals known to us, then it is likely that the ORF is part of a gene.

5.6 Experiments

This section details two experiments we have performed to evaluate our method. The first experiment shows that our method is promising as it has performance similar to that of MetaGene, a recently developed metagenomics gene prediction
method. The second experiment shows the prediction accuracy on a richer simulated data set with fragment lengths normally distributed about a mean.

The results of the first experiment serve only as a guide, rather than a definitive benchmark on performance differences. One reason for this is that the data used for training and testing are only similar in microbial diversity and fragment length to the one used in the evaluation of MetaGene. The exact fragments used for constructing and evaluating MetaGene are not publicly available. We also differ from MetaGene in the units of classification. Our classification problem focuses on predicting the class membership of marker codons. In contrast, the inexact gene matching experiment used to evaluate MetaGene are based on ORF segments that can potentially contain several of our marker codons.

In an inexact gene matching experiment, a prediction is correct if there is sufficient overlap between an annotated gene and the predicted ORF. Although the fundamental units of classification is different, our experiments are similar in nature and provide an approximate comparison between the two approaches. We remain focused on our model of classification because metagenomics fragments are usually plagued with sequencing errors. In a realistic data set, it would be difficult to extract meaningful ORF candidates. Our unit of classification is potentially more flexible because a simple extension that is applicable to erroneous fragments involve taking random codons as markers for analysis.

5.6.1 Data Set and Parameters for MetaGene Comparison

In this experiment, we use the 116 bacteria listed in the supplementary material of MetaGene for training. For evaluation, we use the same 10 bacteria genomes listed in the MetaGene paper [67]. We simulate a metagenomics data set by extracting random fragments of 700 bps, at a sample rate of 10% for each genome. The equal sampling across genomes represents a high complexity metagenomics data set where no species is over-represented. We use all curated protein coding regions in the training set for constructing the codon usage and protein length
models. To find clusters of codon usage tables, we use $k = 9$ after tuning via cross-validation on the training set. Context tree models are built using protein fragments found on the simulated metagenomics training fragments. Finally, we also performed an experiment where we over-sample the under-represented positive set by duplicating each positive example. This is a typical trick used to simulate asymmetric costs when there is an imbalance in the size of positive and negative set. Similar to the MetaGene setup, all minuscule uninformative genes less than 60 bps in length are excluded.

5.6.2 Data Set and Parameters for Simulated Metagenome

In this experiment, we randomly selected training and test genomes from the NCBI database (70 for train, 30 for test). We simulate a metagenomics data set by extracting random fragments with length normally distributed around the mean of 1,000 bps and a standard deviation of 100 bps. Each genome is sampled at 10%. We use all curated protein coding regions in the training set for constructing the codon usage and protein length models. To find clusters of codon usage tables, we use $k = 9$ after tuning via cross-validation on the training set. Context tree models are built using protein fragments found on the simulated metagenomics training fragments. Given that there is ample amount of microbial data available from the NCBI Microbes Resources, we believe that this experiment provides a representative evaluation of our method.

5.7 Results

We use 100 iterations of LogitBoost (a variant of AdaBoost) to build each of our codon classifiers. Figure 5.6 shows the results of running our classifiers on a simulated data set with similar microbial composition to the one used in MetaGene evaluation. The figure illustrates that our method has performance similar to that of MetaGene. On average, our best classifier achieves average sensitivity of 92.1% and specificity of 85.6% while MetaGene achieves average sensitivity of 94.0%
and specificity of 89.1%. This particular classifier doubles the size of the positive training set to simulate asymmetric costs. Table 5.1 shows the results on running our classifier on another simulated metagenomics data set.

5.8 Conclusion

At the time of writing, gene finding on metagenomics sequences is a relatively new problem with few solutions available. Traditional prokaryotic gene finders, intrinsic or extrinsic, are not applicable to the problem as they are designed to be used on specific microbes with complete genomic information known a priori. In a typical metagenomics setting, the concept of species is likely undefined and the diversity of microbes makes it extremely costly to sequence sufficient genomic information for meaningful fragment assembly. This is problematic for current prokaryotic gene finding standards.
We have presented a novel method that approaches this new computational problem using boosting. Our method is species independent, and the current results suggest that it is comparable to the best method currently available. It does not require fragment assembly for pinpointing probable gene coding regions and is readily extensible to incorporate any score-based intrinsic and extrinsic descriptors. Our initial results also show that our method can benefit from oversampling positive training data to account for the imbalance in class sizes. This suggests that using asymmetric forms of boosting may result in even more accurate codon classifiers. We plan to pursue this line of work in the near future.

Table 5.1: **Experiment with simulated metagenomics data set.**

<table>
<thead>
<tr>
<th>bacteria group</th>
<th>sensitivity</th>
<th>specificity</th>
<th># pos.</th>
<th># neg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actinobacteria</td>
<td>0.922985</td>
<td>0.896752</td>
<td>1675</td>
<td>13049</td>
</tr>
<tr>
<td>Alphaproteobacteria</td>
<td>0.915888</td>
<td>0.879910</td>
<td>856</td>
<td>6999</td>
</tr>
<tr>
<td>Bacteroidetes/Chlorobi</td>
<td>0.931034</td>
<td>0.876106</td>
<td>319</td>
<td>3741</td>
</tr>
<tr>
<td>Betaproteobacteria</td>
<td>0.883721</td>
<td>0.853933</td>
<td>172</td>
<td>1674</td>
</tr>
<tr>
<td>Cyanobacteria</td>
<td>0.877712</td>
<td>0.923237</td>
<td>507</td>
<td>3995</td>
</tr>
<tr>
<td>Deinococcus-Thermus</td>
<td>0.961538</td>
<td>0.931677</td>
<td>156</td>
<td>822</td>
</tr>
<tr>
<td>Firmicutes</td>
<td>0.928969</td>
<td>0.865391</td>
<td>2872</td>
<td>19843</td>
</tr>
<tr>
<td>Gammaproteobacteria</td>
<td>0.934729</td>
<td>0.901961</td>
<td>1624</td>
<td>14275</td>
</tr>
<tr>
<td>Planctomycetes</td>
<td>0.857143</td>
<td>0.935593</td>
<td>322</td>
<td>4963</td>
</tr>
<tr>
<td>Spirochaetes</td>
<td>0.893536</td>
<td>0.900383</td>
<td>263</td>
<td>2096</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.91073</strong></td>
<td><strong>0.89649</strong></td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Bibliography


