I give permission for public access to my thesis and for any copying to be done at the discretion of the archives librarian and/or the College librarian.

_____ _____

Signature Date

Abstract

Proteins are one of the most important biological structures found in nature. Consequently, the ability to determine a protein's function quickly and accurately is of considerable importance to the scientific community. Many modern computational techniques seek to solve this problem by using available structural information to classify proteins in order to infer protein function.

Most current protein classification systems use a combination of automated techniques and manual curation. Even so, some levels for certain systems rely solely on expert analysis. Generally, in these circumstances, the characteristics of the grouping are considered too broad or vague for automated techniques. The **architecture** level of CATH is one such example.

This thesis explores the application of rigidity theory to the physical structure of proteins in **automated protein classification** by augmenting standard protein data (in the form of secondary structure information) with rigidity data. **Machine learning** algorithms are used to perform the classification.

We evaluate the effect of the rigidity data on the accuracy of these algorithms. Specifically, we focus on three architectures within the Mainly Alpha class: orthogonal bundle, up-down bundle, and alpha/alpha barrel.

Automated Protein Classification Using Rigidity Analysis

Courtney Schirf

Professor Audrey St. John, Faculty Advisor

Presented to the faculty of Mount Holyoke College in partial fulfilment of the requirements for the degree of Bachelor of Arts with Honours

May 2011

Acknowledgements

First, I would like to thank my wonderful advisor, Audrey, for her support and direction, not only as my thesis advisor, but also as my academic advisor, and for introducing me to the field of rigidity theory, of which until recently I was wholly unaware and with which I find myself now greatly intrigued.

I wish to thank my parents for having inculcated me with a strong sense of curiosity and a love for natural sciences, and for teaching me never to shy away from things technical or difficult.

Finally, I would like to thank all of my family and friends for their unconditional support and for their willingness to help proofread and edit the thesis presented herein, even when the subject material was not always scintillating.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Proteins are one of the most vital and versatile biological structures found in nature. They range from the incredibly complex, like hemoglobin, which is composed of four sub-units and which carries oxygen in red blood cells, to the simple, like kyotorphin, a dipeptide which facilitates pain regulation in the brain [45].

In every case, they play a crucial role in all aspects of our day-to-day existence, from the membrane proteins that transport calcium and potassium into cells to the polymerases which replicate DNA and RNA to the antibodies that fight infections; and, in every case, the protein's physical shape, or *conformation*, determines its function. An example of a protein's structure is given in Figure 1.1.

In light of this, it is not surprising that currently one of the important areas of biology and bioinformatics is inferring protein function from data such as sequence and structural information.[1] This is particularly important for researchers who have discovered a novel protein and are trying to determine its function and for researchers trying to design proteins for drug therapy.

It is understood that proteins with similar shapes often perform similar functions, and current techniques are using available structural data to infer protein function (indeed, the entire field of structural bioinformatics is dedicated to the analysis

[1]The reader who is less familiar with the biology behind protein formation is encouraged to refer to Section 2.1, where this topic is treated in more detail.

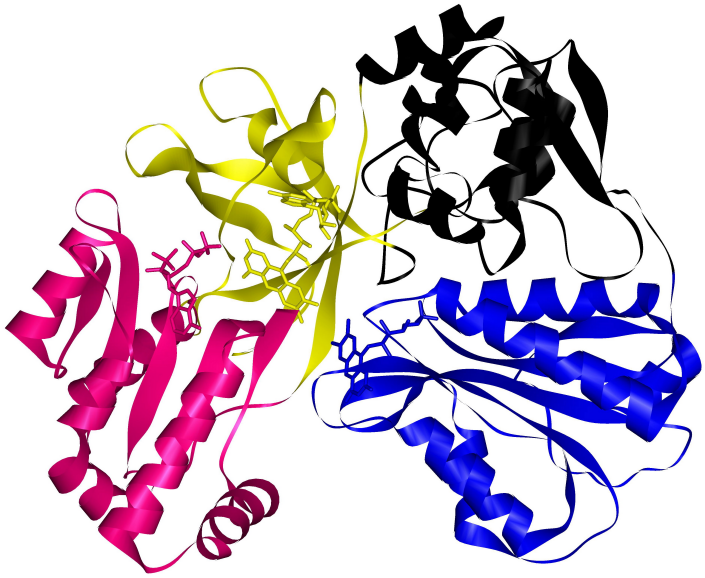


Figure 1.1: A ribbon model of example protein human cytochrome p450 reductase, courtesy of Dr. Christopher Maronic, Dr. Bettie Sue Master's Lab, University of Texas Health Science Center at San Antonio

of the structure–function relationship in macromolecules) [6, 20]. These systems classify proteins based on different kinds of similarity, so that the structure of a new protein can be compared against the structures of known proteins. After the new protein is classified, we can infer that its function is similar to the function of the other proteins in the same class (or other level of the appropriate hierarchy).

1.1 Problem statement

Assigning each protein to the appropriate class is a slow and labour–intensive process. It often requires a human researcher to carefully analyse a protein's structure then classify the protein manually.

However, there is an abundance of raw data for proteins whose structures have been resolved. Now, the problem is to design systems to compare new protein structures against known ones in order to classify them. These systems often take a hierarchical form, where proteins grouped at lower levels of the hierarchy show a greater level of similarity than proteins grouped at the top levels of the hierarchy.

Creating methods to automate this procedure is the problem of **automated protein classification**. These methods significantly speed up protein classification, which is of interest to the scientific community in light of not only the sheer amount of data currently available, but the rate at which such information is growing. For example, the CATH database (to be discussed in more detail in Sections 1.2 and 2.2) contained 86,151 domains in 2006. The 2010 release contained 152,920 domains.

Many current protein hierarchies use automated procedures at lower levels of the hierarchy. However, some still rely on manual inspection for the higher levels, which are considered too broad and subjective for current automation techniques [39].

1.2 Related Work

In this section, we give a brief overview of related work in the areas of automated protein classification and rigidity analysis, and we describe some of the machine learning algorithms used. More detailed background information is given in Chapter 2.

1.2.1 Automated Protein Classification

The field of automated protein classification has been studied extensively since the early 1990s. For a general survey of some of the major systems, see [23]. Principally, most of the systems in this area are concerned with classifying proteins by secondary structure. The two most popular structurally-based protein classification systems, CATH and SCOP, are primarily manual systems that use a few automated methods [20, 39, 33].

These systems classify proteins according to pre-defined classes. It is then possible that a classification scheme based on the data alone may reveal connections between some proteins that would not have been seen using the current classification methods.

Both CATH and SCOP are hierarchical systems, where each level of the hierarchy is pre-determined. The CATH database is based on *domain* similarity, while the SCOP database is built on evolutionary relationships (mainly from sequence data), although domain information is also used [20]. Domains are compact areas of a protein that are capable of folding and functioning independently of all other areas.

The SCOP (**s**tructural **c**lassification **o**f **p**roteins) database organises proteins in a hierarchy in the following order: classes, folds, superfamilies, and families [33]. Figure 1.2 shows how the SCOP database is organised, using the domain cytochrome p450 2c8 as an example.

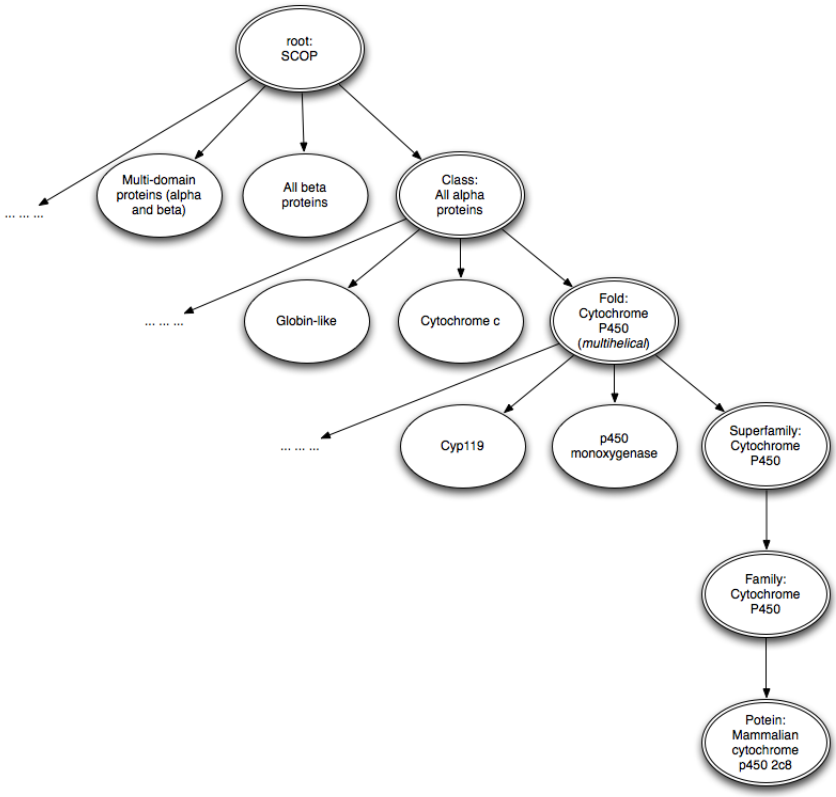


Figure 1.2: SCOP hierarchy for example domain human cytochrome p450 2c8

The CATH database is named after the organisation system it uses: **c**lass, **a**rchitecture, **t**opology (or fold), and **h**omologous superfamily. An example of the CATH hierarchy is given in Figure 1.3.

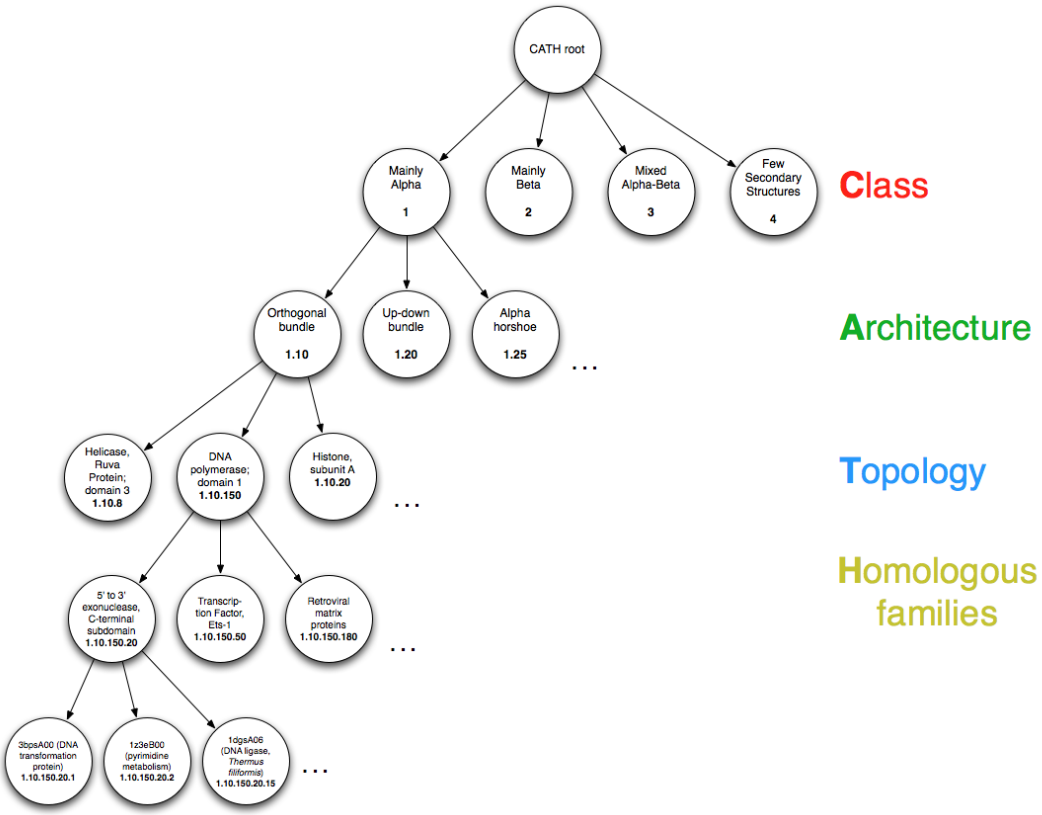


Figure 1.3: Example of the CATH hierarchy

1.2.2 Machine Learning in Automated Protein Classification

The application of machine learning to biological problems is a well-established approach in the field of bioinformatics, particularly for protein fold recognition and protein classification [52, 21]. Such techniques have been used for general problems, including predicting protein structure [27] or classifying proteins [7], to more specific problems, such as identifying key amino acid positions in pathological immunoglobin-type beta domains [53].

1.2.3 Rigidity Analysis of Proteins

In general, rigidity theory is concerned with analysing rigid and flexible regions of an object. It has recently been applied to proteins. One such example is the FIRST software from Arizona State University [28, 14].

To date, FIRST has been used to identify rigid regions of proteins (namely, hemoglobin) for the purposes of drug design [8], identifying a protein folding core [24], and analysing the protein-protein complex formed H-Ras and C-Raf1 (types of kinases) [18].

Section 2.3 discusses how FIRST represents proteins and theory behind the analysis.

1.3 Contributions

The work of this thesis is aimed at improving the current methodology for automated protein classification by augmenting current standard protein quantizations with data garnered from rigidity analysis. Our results show that the inclusion of even basic rigidity information improved the accuracy of several machine learning techniques on a difficult dataset.

Because rigidity theory has only recently been employed for protein structure analysis, it is possible, given the promising nature of our results, that such a technique will become a common tool for protein classification.

1.4 Structure of thesis

The following chapters give some background into the problem of automated protein classification and rigidity analysis, discuss the approach used in this study, and provide a discussion of the results.

Chapter 2 gives an overview of the core biological concepts referenced throughout the thesis and of the machine learning techniques used. Chapter 3 describes the methodology used in choosing a hierarchy to work with, in gathering the requisite data, and in performing the classification. Chapter 4 presents our results; we then discuss the conclusions drawn from these results in Chapter 5 and propose some future areas of research.

Sample Python and bash scripts may be found in the appendices, along with a

brief tutorial on how to use these scripts in order to perform the same analysis on a different set of proteins.

Chapter 2

Background

In this chapter we introduce some basic protein biology concepts that are necessary to understand this study; we discuss some details of the the specific protein hierarchy (CATH) within which we worked; we give an overview of the theory behind the rigidity analysis we use; and, we give short descriptions of the machine learning techniques used to preform the classification.

2.1 Protein Biology

For the reader unfamiliar with the relevant biology concepts behind protein formation, we provide a brief overview of these ideas here. The central dogma of protein formation is DNA $\longrightarrow$ RNA $\longrightarrow$ protein,[1] meaning that DNA is transcribed to RNA which is translated into a sequence of amino acids. Each amino acid is connected to the other by a peptide bond. This *primary structure* of a protein is often termed "beads–on–a–string," where each amino acid is a "bead" [6].

Depending on the specific properties of each amino acid and those of its neighbours,[2] the string will bend to form the protein's *secondary structure*, mainly α-helices (Figure 2.2) and β-pleated sheets (Figure 2.3). Because the formation of

[1]Technically, the last step should read 'polypeptide,' as the term 'protein' implies a tertiary structure, but this distinction is not always maintained, nor will it be here. For the purposes of this paper, the terms protein and polypeptide will be used interchangeably.

[2]Specifically, the placement of hydrogen atoms.

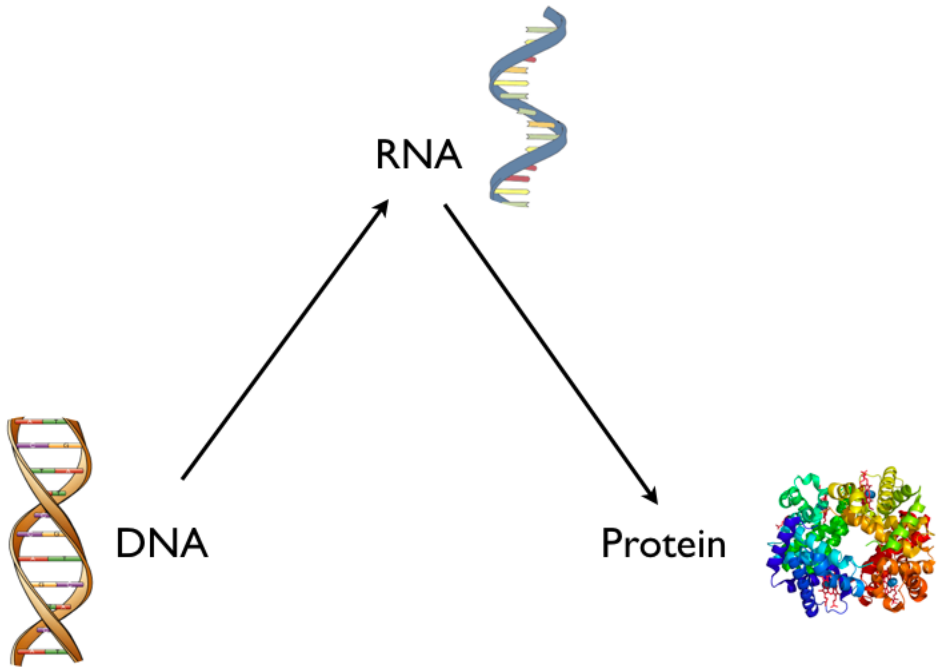


Figure 2.1: Central dogma of protein formation

these structures is dependent only on neighbouring groups of amino acids (i.e., because these are local structures), both α-helices and β-pleated sheets can exist in a single protein structure [6, 20].

Some amino acids[3] then form peptide bonds between the secondary structures, giving a protein its *tertiary structure*, also called a protein's *fold*.[4]

Proteins are comprised of *domains*: compact, stable regions of a protein that fold and function semi-independently of the rest of the protein. It should be noted that domain determination is normally a manual and subjective process. At the moment, the research group that resolves a protein structure usually describes which areas are domains; then other groups reference this literature.

This subjectivity arises from the lack of a single, universally accepted definition for what comprises a domain; the most common definition is that given above [42,

[3]Primarily methionine and cysteine, since these contain sulfur, which is necessary for the formation of disulfide bridges.

[4]Although the formation primary, secondary, and tertiary structures are often thought of (and taught as) an ordered process, it should be noted that these steps often occur in tandem in the cell.

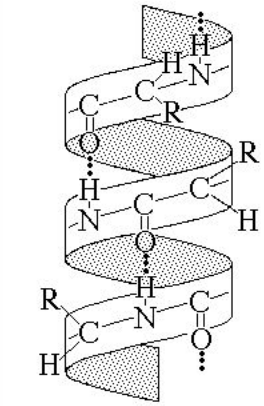


Figure 2.2: An example α helix (J. José Bonner)

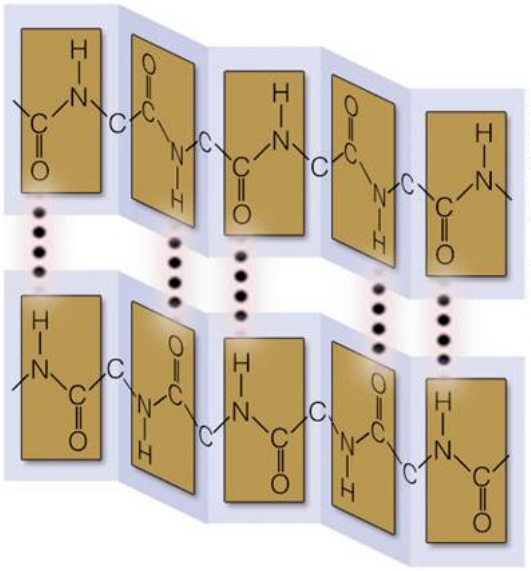


Figure 2.3: An example β-pleated sheet (Brooks/Cole)

43]. Despite this, starting in the late 1970s, work has been done to automate domain determination [9, 43, 1].

Protein domains are comprised of multiple *motifs*, three-dimensional structures that occur in multiple proteins. Usually, these are combinations of secondary structures, but sometimes certain combinations of amino acids can also be considered motifs. Some common motifs include the helix–turn–helix motif (Figure 2.4) and the four–helix bundle [6, 20].

Each of these structures can be useful for classifying proteins. Domains in particular are usually conserved among evolutionarily related proteins [20]. Thus, protein classification is primarily done by a secondary structure grouping, domain similarity, and motif similarity.

Sequence data has been used traditionally for this purpose, but it has some limitations. First, nucleic acid sequences can be similar by chance, especially if the sequences are short. Second, not all parts of a nucleic acid sequence are used to create proteins. This is primarily true of DNA, where certain sequences called introns are removed in the transcribed RNA. Nor is it always clear which genes

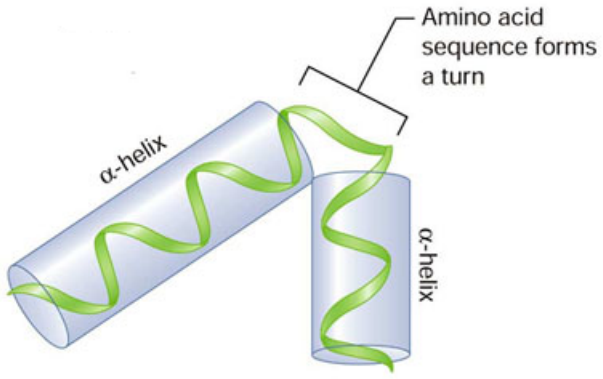


Figure 2.4: Generalised form of a helix–turn–helix motif (Scott Freeman's *Biological Science*, Prentice Hall)

encode which proteins [20].

Structural data, on the other hand, is more difficult to obtain. Although the actual methods by which this data is obtained is beyond the scope of this project, it is worth saying a few words about them to put this study in context. The main difficulty in determining a protein's structure is simply that proteins are too small to be directly seen using any technology currently available [6]. Consequently, indirect methods are used, but these have their drawbacks. Crystallography, for example, forces proteins, which are not (completely) rigid, into a rigid conformation. This can, and often does, produce small conformational changes in a protein's side chains or change the orientation of some molecules in the structure [20]. Moreover, it is extremely expensive and time-consuming to use purely biological methods to determine protein structure and function. Whole wet labs often spend several years resolving the structure of a single protein.

2.2 The CATH System

The main goal of this project is to evaluate the effect of rigidity data on the accuracy of machine learning techniques when classifying proteins at one of the higher, more subjective hierarchy levels [39]. Specifically, we are interested in the **architecture**

level of CATH, so we will discuss this particular hierarchy in more detail.

Unlike SCOP, CATH is a more structurally based classification system. The motivation for focusing on structural information rather than sequence information is that proteins within families of structurally related proteins (which have a high likelihood of performing similar functions) may have low sequence similarity [39].

CATH has four main hierarchy levels, from which its name is derived: **C**lass, **A**rchitecture, **T**opology, and **H**omologous families. The reader is encouraged to look back at Figure 1.3 for a graphical depiction of the general structure of CATH.

To date, CATH contains 1,282 folds and 152,920 domains. Each of these domains are placed into a unique classification using a mix of automated and manual methods.

The **class** level is the highest level in the hierarchy, and it refers to the secondary structure composition of the domains (*e.g.*, the relative number of α-helices and β-sheets). CATH uses 4 classes: Mainly Alpha, Mainly Beta, Mixed Alpha–Beta, and Few Secondary Structures. As their names suggest, the Mainly Alpha and Mainly Beta classes contain mostly α-helices or β-sheets, respectively. The Mixed Alpha–Beta class contains domains that do not have predominately more α-helices or β-sheets. Some other systems distinguish Alpha/Beta (α/β) from Alpha + Beta proteins. The former are distinguished as having alternating α-helices and β-sheets, while in the latter, these structures occur in different areas of the protein.

A well-cited example of an α/β protein is triose phosphate isomerase, shown in Figure 2.5 below. Figure 2.6 shows ribonuclease A, an example of an $\alpha + \beta$ protein.

The Few Secondary Structures class contains proteins that do not have sufficient secondary structure information for a classification into any one of the other three classes. This class has the smallest population of proteins [39].

Classification at the **class** level is a mix of automated and manual procedures, where an algorithm is applied that does a first pass at the domains [34]. This algorithm gives an accuracy of about 90%. Then the proteins are reviewed manually for the final classification [39, 10, 50].

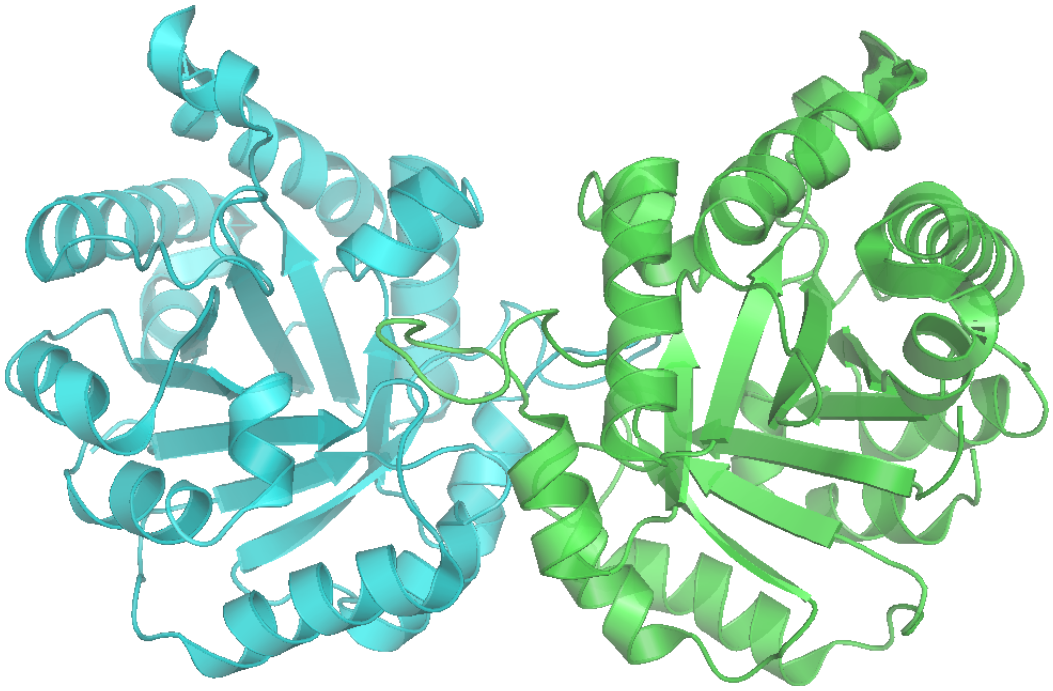


Figure 2.5: Triose phosphate isomerase, an example of an alternating α/β protein (Richard Wheeler, 2006)

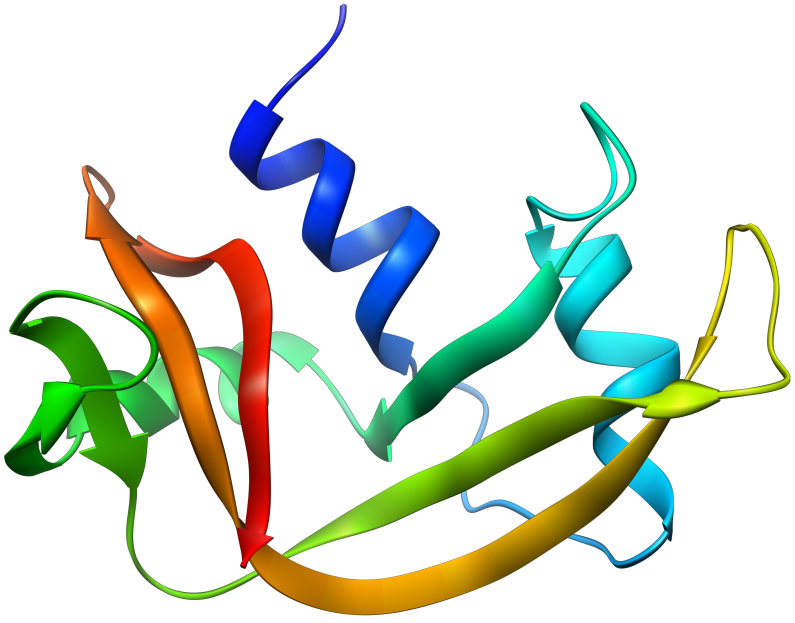


Figure 2.6: Ribonuclease A, an example of an $\alpha + \beta$ protein (The Full Wiki)

The **architecture** level is the next level after class, and it describes general features of the fold shape and the orientation of the secondary structures [50, 39].

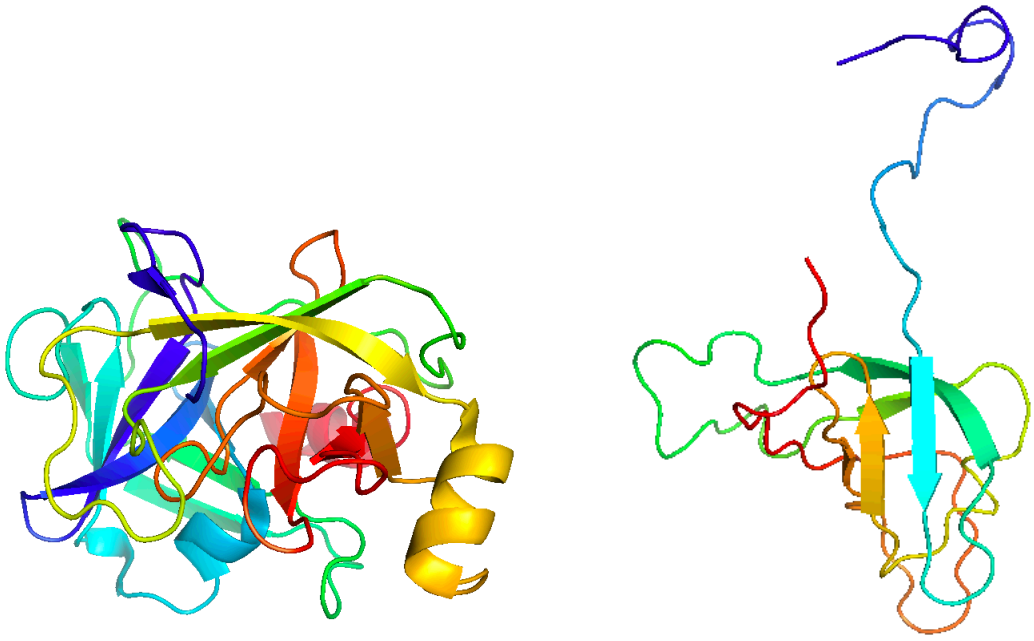
This does not take into account the links between the secondary structures. One example of an architecture is a β-barrel, such as in the protein structure in Figure 2.7. Other architectures include $\beta - \alpha - \beta$ sandwiches and jelly rolls. Classification at this level is done entirely manually using standard classification descriptions; there is no automation procedure [39, 41, 11, 10]. Domains for which the secondary structure arrangement is particularly complex and cannot be classified into known categories are placed in a general 'complex' architecture [39].

Figure 2.7: A monomer of a sucrose-specific porin, an example of a β barrel protein (The Full Wiki)

The next level is **topology**, in which proteins are grouped by fold similarity (*i.e.*, they have a similar arrangement of secondary structures and similar connections between the secondary structures).[5] Examples of topologies include porin and thrombin, subunit H, which both fall under the beta barrel architecture in the Mainly Beta class.

The fourth level is **homologous families**. Proteins grouped at this level have high structural and functional similarity. For example, two topologies within the thrombin, subunit H topology are trypsin-like serine proteins and phage tail proteins. Examples of both kinds of proteins are shown in Figure 2.8 below.

[5]In CATH, the term topology is used to mean fold.



(a) Trypsin-like serine protein (b) Phage tail protein

Figure 2.8: Examples of the thrombin, subunit H topology (beta barrel architecture, Mainly Beta class)

Classification done at the topology and homologous family levels relies heavily on automation techniques. The main source of automation is a sequence structure alignment program (SSAP), a technique which has been greatly expanded by the CATH group [38]. Domains are first scored on a similarity scale from 0 to 100 by a fast SSAP. Those that have a score of less than 75 are run using a more sensitive (and also slower) version of the SSAP algorithm. At the end of this process, domains that have a SSAP score of 70 or higher are put in the same topology. Domains are assigned to homologous families using a combination of a SSAP score of 80 or higher and high functional similarity determined by referencing to the literature [39].

The CATH hierarchy has recently added 5 more levels called **sequence family levels**, the names of which make up the acronym SOLID. Proteins found together in any one of these levels have $> 35\%$ sequence similarity. Each level after S up to I indicates a higher degree of sequence identity, as seen in Table 2.1 below. The D

Level	Sequence Identity	Overlap
S	35%	80%
O	60%	80%
L	95%	80%
I	100%	80%

Table 2.1: CATH sequence family levels with corresponding sequence similarity thresholds

level of SOLID is only to give each domain a unique classification [39, 41, 48].

2.3 Rigidity Theory

Rigidity theory provides a rigorous mathematical foundation for efficiently analysing rigid properties of structures. We are interested in combinatorial rigidity, which captures rigidity properties by abstracting a structure to an associated graph.

We begin with a description of some of the main types of structures that exist. The simplest is the *bar-and-joint*. *Joints* can be thought of as points or pins. These joints have *degrees of freedom*, which intuitively can be thought of as the number of constraints necessary to rigidify the structure. In 2-dimensional space, joints have 2 degrees of freedom; in 3-dimensional space, they have 3 degrees of freedom.

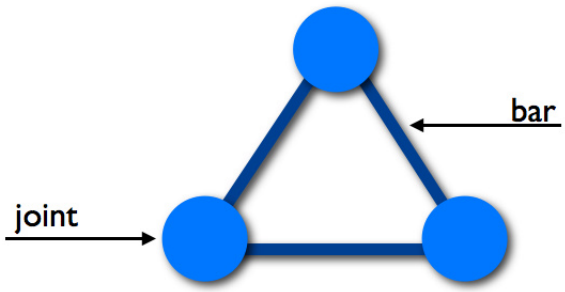


Figure 2.9: An example of a rigid bar-and-joint structure with 3 bars and 3 joints

Bars connect the joints. They are a kind of *constraint* on the system, because they limit how a joint can move. Specifically, they remove 1 degree of freedom (both in 2- and 3-dimensional space). An example of a simple bar-and-joint structure with

3 joints and 3 bars is given in Figure 2.9.

Body-and-bar structures are similar. Here, bodies can be thought of as rigid structures. In 2-dimensional space, they have 3 degrees of freedom; in 3-dimensional space, they have 6 degrees of freedom. With this structure, we can introduce another constraint called a *hinge*, which leaves 1 degree of freedom in both 2- and 3-dimensional space (*i.e.*, it removes 2 degrees of freedom in 2-dimensional space and 5 degrees of freedom in 3-dimensional space). An example of a *body-bar-hinge* structure is given in Figure 2.10. In this figure, the purple "blocks" are bodies, the black lines with circular endpoints are bars, and hinges are represented by the black lines between the purple "blocks." Since a hinge may be represented by 5 bars [47], we focus our discussion on bar constraints.

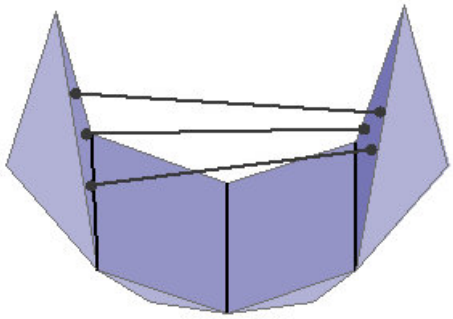


Figure 2.10: An example of a body-bar-hinge structure

We now formalise the structures described. In all cases, we abstract the structure to a graph, $G = (V, E)$, where V is a set of vertices and E is a set of edges. In the *bar-and-joint* structure, each joint is a vertex in G and each bar is an edge. In the *body-and-bar* structure, each body is associated with a vertex and each bar with an edge.

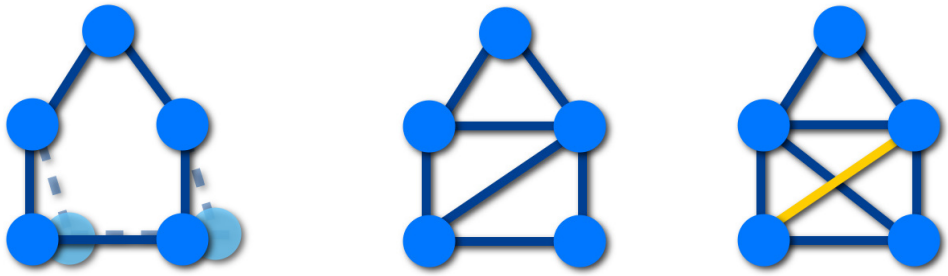
Let us now associate a distance function, $\ell : E \longrightarrow \mathbb{R}$, that assigns each edge a fixed length. A graph and distance function is called a *framework*.

Now that each edge has a fixed length, it is possible to attach coordinates to each vertex in some real space $\mathbb{R}^d$, thereby "anchoring" the structure to a particular form. The coordinates given are called an *embedding* (also called an *injection* or a

realisation).

It is easy to see that a single framework could have multiple embeddings. For example, consider a translation of the G in $\mathbb{R}^d$ space. There are an infinite number of possible translations; therefore, there are must also be an infinite number of possible embeddings [19]. We are interested in distinct embeddings, *i.e.*, embeddings that are not a result of translations or rotations.

We are now in a position to ask: is the embedding rigid? To answer this question, imagine pushing on the structure in Figure 2.9 (without changing the edge lengths). No amount of pushing will *deform* (change the shape of) the structure. Therefore, Figure 2.9 is rigid.[6]



(a) (b) (c)

Figure 2.11: (a) A flexible pentagon-shaped graph; (b) A minimally rigid pentagon-shaped graph; (c) A rigid pentagon-shaped graph with one overconstraint (shown in gold)

Now consider the structure in Figure 2.11a. If we pushed on that structure, we could deform it. However, if two bars are added as shown in Figure 2.11b, we can no longer deform the structure; it is rigid. Consider adding yet another bar as in Figure 2.11c. Such a bar is an *overconstraint* because the rigidity information it adds is redundant: it provides the same rigidity information as the bar added in Figure 2.11b.

A structure that contains overconstraints such that it remains rigid if any bar is removed is considered *redundantly rigid*. A rigid structure that contains no overcon-

[6] In fact, it is *globally rigid*, but we will not make this distinction here.

straints is called *minimally rigid*. The counting conditions presented in Theorem 2.3.1 define when a structure is *generically minimally rigid* in the plane [31]. A full discussion of genericity is not within the scope of this thesis, but, intuitively, a generic embedding of a structure is one that somehow encompasses the characteristics of most embeddings for that structure; or, put another way, there is nothing special about the embedding.

Theorem 2.3.1 (G. Laman, 1970) *A graph (V,E) is generically minimally rigid for dimension 2 if and only if:*

1. $m = 2n - 3$, *where* m *is the total number of edges in the structure and* n *is the total number of vertices*
2. *For all subsets of* n' *vertices where* $n' \leq n$, $m' \leq 2n' - 3$, *where* m' *is the number of edges spanned by the* n' *vertices*

A pair of theorems gives us a set of counting properties that define when a body-and-bar structure is rigid in 3-dimensional space. First, Theorem 2.3.2 defines when a body-and-bar structure is rigid in 3-dimensional space (no counting conditions yet) [46].

Theorem 2.3.2 (Tay, 1984) *Given a 3-dimensional body-and-bar structure, associate a graph by mapping bodies to vertices and bars to edges. The original structure is rigid if and only if the associated graph is the edge-disjoint union of 6 spanning trees.*

The characterisation of graphs with edge-disjoint union of k spanning trees in Theorem 2.3.3 gives us the counting properties for rigid body-and-bar structures in 3-dimensional space [37, 49].

Theorem 2.3.3 (Nash-Williams and Tutte, 1961) *A graph with* m *edges and* n *vertices is the edge-disjoint union of* k *spanning trees if and only if:*

1. $m = kn - k$
2. *for all subsets of* n' *vertices,* $m' \leq kn' - k$*, where* m' *is the number of edges spanned by the vertex set*

Even if a whole structure is not rigid, it may be possible that parts of that structure are. Such parts are called *rigid components*. An example of such a structure is shown in Figure 2.12, where the triangular part of the structure (shown in gold) is rigid, while the square portion is not. The triangle, then, is a rigid component of the structure.

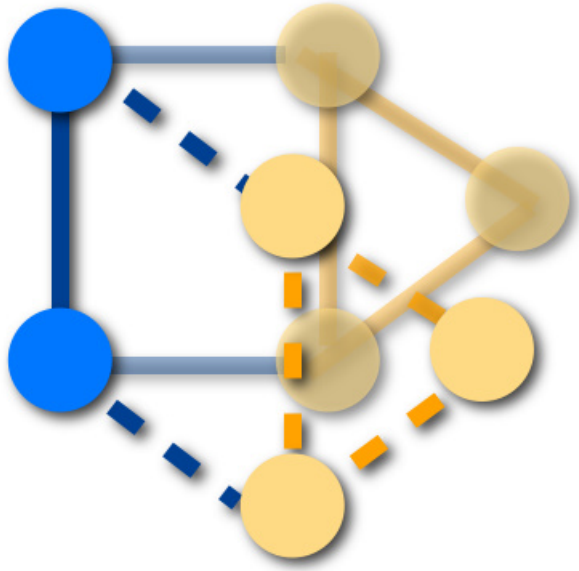


Figure 2.12: An example of a flexible structure with a rigid component (shown in gold)

In 3-dimensional space, the bar-and-joint model is not well understood, and is, in fact, one of the biggest open questions in rigidity theory. Since the body-bar-hinge model is well-understood in 3-dimensional space, it is the model used by the rigidity analysis tool, FIRST (Floppy Inclusions and Rigid Substructure Topography) from Flexweb, which is freely available for academic purposes. In this model, each atom in the protein is a body. Strong kinds of bonds (*e.g.*, covalent bonds) are represented

by hinges. Weaker bonds (*e.g.*, hydrogen bonds) are represented as bars.

FIRST analyses this model using a version of the *pebble game* (briefly described below), looking for rigid components in the structure. These rigid components are what are returned by the program. Furthermore, FIRST is able to identify regions that contain overconstraints. FIRST calls these areas *stressed regions*. This concept is important in its application to proteins. For example, hydrogen bonds break and reform many times over in the same structure. If overconstraints did not exist in this system (*i.e.*, the protein), the loss of a hydrogen bond would result in the collapse of the entire structure.

We give an overview of a basic pebble game here to familiarise the reader with the general idea of the theory behind FIRST's analysis. More detailed information about the pebble game in general and FIRST's 3-dimensional implementation can be found in [32, 28, 29].

The pebble game is an algorithm used to analyse the rigidity of a structure. It uses a graph, G, as its input, along with the parameters ℓ, where $\ell + 1$ is the minimum bound on the number of pebbles (discussed further below), and k, which is the initial number of pebbles on each vertex and the maximum number of pebbles that may be present on a vertex at any given time. A graph, D, is created, which contains no edges but contains all of the vertices in G with k pebbles on each vertex.

When the game is played, edges from G are added to D if and only if the endpoints, u and v, of that edge have a total of $\ell + 1$ pebbles between them. If the endpoints do not share enough pebbles, the game is allowed to search for them according to certain rules. Each time an edge is added into D, a pebble is removed from u.

After the game is over, we can use information on how many pebbles are left in the game and whether or not an edge was rejected to infer properties of the original structure. For example, if exactly ℓ pebbles remain in the game and no edges were rejected, then the structure is minimally rigid. If, however, ℓ pebbles are left in the game, but at least one edge was rejected, then the original structure contained at

least one overconstraint. Furthermore, if more than ℓ pebbles are left, those pebbles correspond to the degrees of freedom left in the structure [32].

The exact information gathered from FIRST is discussed in Section 3.2.

2.4 Machine Learning

This project relies on machine learning, a subset of artificial intelligence concerned with discovering rules from a dataset. This discipline is often used for classification and clustering problems. In *supervised* machine learning, labelled training data is sent as input and the algorithm generalises a model from the labels (how this step is done is specific to the particular algorithm used). Then, unlabelled testing data is sent as input and the algorithm returns some output, which can be tested to determine the accuracy of the algorithm [13, 30, 35]. Diagrams of this process are presented in Figures 2.13 and 2.14.

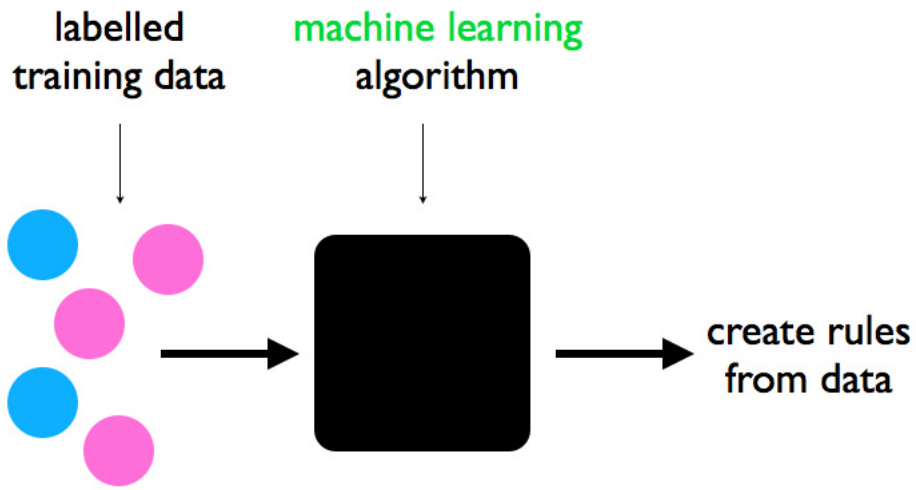


Figure 2.13: Machine learning diagram, training the algorithm

There are four main supervised machine learning algorithms that will be considered for this project: Naïve Bayes, support vector machines (SVM), bootstrap aggregating (bagging), and Adaptive Boosting (AdaBoost). Each will be used at black–box level.

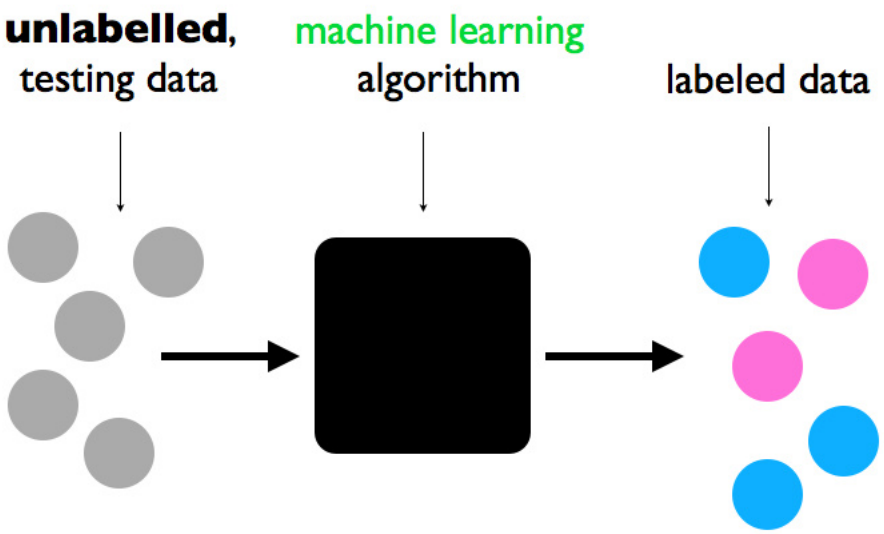


Figure 2.14: Machine learning diagram, running the test set

Here we will give a brief overview of these four main algorithms. Information on the other algorithms used can be found in the cited literature.

Naïve Bayes : Naïve Bayes is a classic statistical algorithm that uses Bayesian probability models to classify items. The algorithm assumes that the data is *normal*, e.g., continuous variables have Gaussian distributions. Furthermore, the algorithm assumes that the variables are linearly independent [13, 35]. However, this is not usually the case, so Naïve Bayes is often outperformed by other methods [30].

Support vector machines (SVM) : The central idea of SVM is that of a *margin*, either side of a hyperplane. Specifically, this hyperplane separates the two data classes, so similar elements will be on one side of the hyperplane. The problem for the algorithm, then, is to find the largest margin (*i.e.*, to maximise the distance between the hyperplane and the data points).

The exact mathematical model that determines this distance is called a *kernel function*, and the general SVM algorithm sums over the results of the kernel function for each instance. In essence, the kernel function maps the data to

a higher dimensional space (including infinite dimensional space) because the given data may not be linearly separable. By moving to a different space, the hope is to achieve linear separability [5, 30].

Bagging (Bootstrap aggregating) : **B**ootstrap **agg**regat**ing** is meant to improve classification accuracy, reduce variance, and avoid overfitting. It does so by generating multiple versions of a classifier and then using those to develop an aggregated classifier by averaging over the versions.

Experimental evidence suggests that bagging can help improve *unstable* classifiers (classifiers where a small change in the training set changes the model significantly), but can degrade the performance of a stable classifier [3, 4].

Adaptive Boosting (AdaBoost) : AdaBoost is an algorithm, developed by Freund and Schapire in 1996, which "boosts" the performance of a weak algorithm (a method that does not provide a performance much better than random) by successively applying the weak algorithm on different distributions of the training data [15]. This iteration is stopped when some error rate is reached or when enough weak classifiers have been constructed. Then the classifiers produced by the weak algorithms are combined into a single composite classifier [15, 2].

One of the benefits of AdaBoost is that it does not tend to overfit data. In theory, this is possible if the number of rounds algorithm is run is too large. However, empirical evidence shows that overfitting is rare even when the algorithm is run for thousands of rounds [16].

In addition to the five techniques described above, we also ran our data though the following machine learning algorithms: attribute selected classifier [22], multiclass classifier [22], decision table [36], and multilayer perceptron (a type of feed-forward neural network) [40].

Chapter 3

Methods

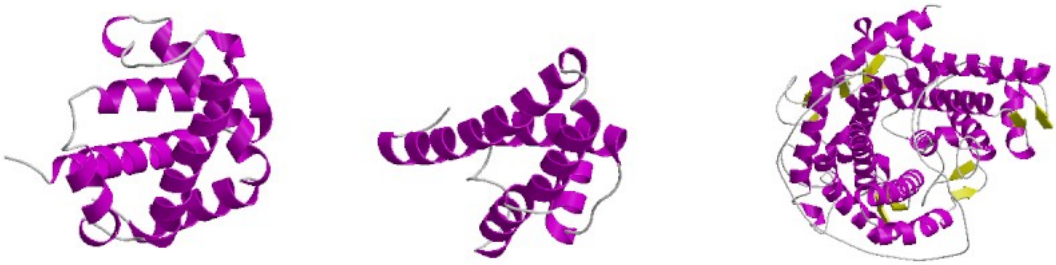
In this chapter, we present the process for gathering and testing data, including how the classification hierarchy was chosen and why rigidity analysis was run on whole proteins instead of domains. The general approach is described first, followed by some implementation details.

We assume that the reader is familiar with the background material presented in Chapter 2, in particular, basic protein biology, especially as it relates to protein structure, and the organisation and procedures used by the CATH hierarchy.

3.1 Choosing a classification hierarchy

An existing classification system was chosen to serve as a benchmark for our results. The two most popular systems at the moment are CATH and SCOP, which are both semi-automated. However, as mentioned in Chapters 1 and 2, the process of classifying domains into **architectures** in CATH is entirely manual [39, 10]. It was for this reason that we chose to work within the CATH framework.

In order to make this project feasible, we decided to focus on a few of the most populous **architectures** in one of the most populous **classes** in CATH. The two most populous **classes** are Mainly Alpha and Mainly Beta, and we arbitrarily chose to focus on the Mainly Alpha class. We then chose the three most populated **archi-**



(a) 2peg (orthogonal bundle) (b) 1k3y (up-down bundle) (c) 1gai (alpha/alpha bundle

Figure 3.1: Example proteins from each of the architectures analysed in this study (orthogonal bundle, up-down bundle, alpha/alpha barrel)

tectures within the Mainly Alpha class: orthogonal bundle, up-down bundle, and alpha/alpha barrel. Examples for sample proteins from each of these architectures are shown in Figures 3.1a, 3.1b, and 3.1c, respectively.

3.2 Choosing the rigidity information

As was discussed in Section 2.1, a single protein may be comprised of multiple domains. Since classification is done at the domain level, the data gathered can either be at the domain level only or at the level of the whole protein. We chose to run the rigidity analysis on the whole protein instead of each domain.

Our reason for doing so is two-fold: (1) while it is possible to specify chains in FIRST, doing so would make this process less easy for batch-processing; and more importantly (2) we are attempting to describe the context in which a given domain occurs. This means that we end up giving the domain slightly more information than we might be able to garner from a new domain alone, but it also adds functional information.

To clarify this reasoning, consider the classification of vehicle parts, such as wheels, doors, axles, etc. While it seems straightforward to obtain data for each part in isolation and classify solely by that information, it may be possible to achieve

better results by taking into account the context in which those parts appear. For example, if a wheel tends to appear with other wheels and an axel and we save this information, then next time we see an object which is in conjunction with wheels and an axel, we will be more likely to classify that object as a wheel.

3.3 An overview of the analysis

We begin with an overview of principal steps of the analysis, which are summarised in Figure 3.2 and described in more detail below.

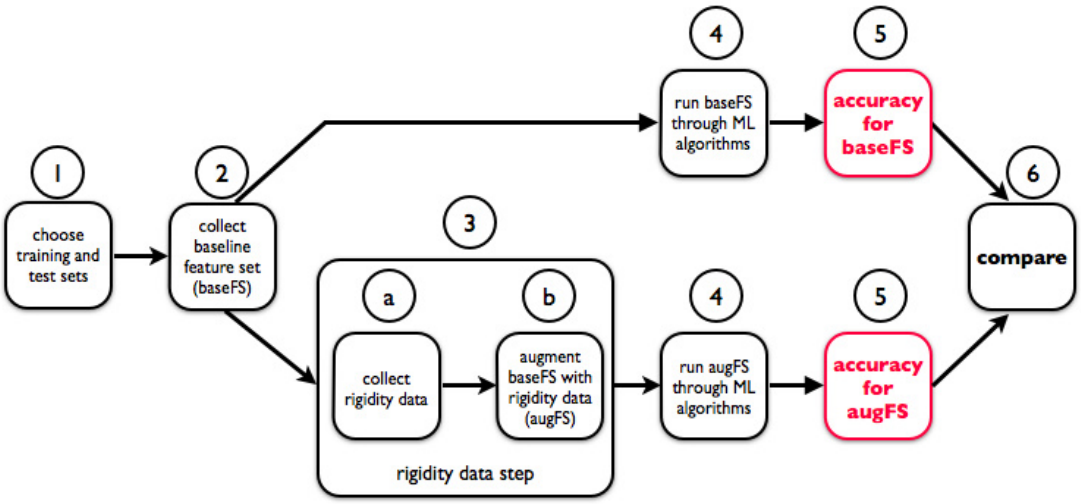


Figure 3.2: Overview of the steps involved in the analysis

Step 1 : The first task is to **choose training and test sets**. This involves obtaining a list of domains whose structures is known and which have been classified in the CATH hierarchy. Naturally, not all domains have been given a classification in CATH, in part because some domains have simply not been added to the system, and it part because some the PDB files do not meet CATH's standards [39].

Step 2 : The **baseline feature set** (baseFS) is created by choosing and quantifying specific characteristics of each domain in the both the testing and training

sets. Possible characteristics include the number of α-helices, the number of β-sheets, or the protein's refractivity.

Step 3 : **Rigidity data** is then gathered on the domains in the testing and training sets by analysing the PDB files using FIRST. The output files from the software are parsed, and the pertinent information is added to the baseline feature set to create the augmented feature set.

Steps 4–6 : The baseline and augmented feature sets are sent to each of the **machine learning algorithms**. Accuracy measurements are obtained, and the scores are compared.

3.4 Implementation details

We now delve a little deeper into some of the implementation details for each of the steps described above.

Readers interested in the details of the scripts and programs used to collect and process the data are encouraged to consult Appendix C, where these topics are given extensive treatment.

3.4.1 Choosing training and test sets

Originally, we gathered an initial baseline feature, which we refer to as the ProtScale set for reasons described later, set containing secondary structure information on semi-randomly selected domains from the aforementioned architectures in the Mainly Alpha class.

The domain selection was semi-random in that domains were selected from the most populous topologies, but also from a wide range of topologies. In order to achieve this, domains were randomly selected from approximately the top 10 topologies from each architecture. Some architectures contain more topologies than others (*e.g.*, the alpha-alpha bundle architecture contains 3 different topologies, while the

orthogonal bundle architecture contains 270 topologies, up-down bundle architecture contains 97 topologies), so the number of topologies chosen was not even among the architectures. A full list of the original domains chosen is given in Table 3.1.

2fmp	1y88	2q0z	1jx4	1wcn	1lb2
2h8f	1pk1	1oxj	1sxd	1b4f	1sv0
1x9x	1eca	1s5y	1pbx	2peg	1ith
3d1k	1cg5	2aa1	1la6	1xg0	1xf6
1qgw	1b8d	1eyx	2c7l	1kn1	1all
1b33	2oyh	1fzg	1jy2	1fzc	1gk6
1czq	1jnm	1gu4	1g2c	1k3y	1hqo
2a2r	2c4j	2ww2	2wvx	1mz9	2spc
1ybz	3fp5	2o3s	1rzh	256b	1mc2
2d29	3bb0	2lis	1wpg	1e91	1lb3
1sj8	1h12	1n1b	1okc	1gxm	1nxc
1rwh	1qaz	5eau	1n7o	1x1i	1hn0
1cb8	1x9d	2ri9	3dss	1w6k	2sqc
1n49	1qqf	1ks8	2ahf	1g9g	1gai
1lf6	1ut9	2p0v	2ww3		

Table 3.1: Domains of the original ProtScale set

Python scripts were written to automate the process of querying the PDB server to obtain PDB[1] and FASTA[2] files for each protein in the set. For about 100 proteins, this process usually took a couple of minutes.

The secondary structure data for this set was obtained from the ProtScale tool on the ExPASy proteomics server (Swiss Institute of Bioinformatics) [17, 44]. Data was collected on 15 properties including secondary structure information and chemical properties using well-known or standard amino acid scales. For a set of about 100 proteins, this process took several hours to a couple of days, depending on the average number of chains for each protein. A full listing of this information and the scales used are given in Table 3.2.

Python and bash scripts were used to automate the process of filling out and submitting the online ProtScale form and to automate the process of collecting and

[1]PDB files contain a protein's structural information in the form of atom and bond information. FIRST requires PDB files because it needs the atom information.

[2]FASTA files contain the amino acid sequence for a protein and are separated by chain. Secondary structure prediction tools usually require FASTA files because they base their predictions on the properties of the protein's amino acid composition.

N$^{o.}$	**Secondary Structure Prediction**	**Amino Acid Scale Used**
1	Molecular weight	standard
2	Polarity	Zimmerman
3	Bulkiness	standard
4	Refractivity	refractivity
5	Hydrophobicity	Eisenberg et al.
6	HPLC/HFBA retention	standard
7	HPLC/TFA retention	standard
8	Percent buried residues	standard
9	Percent accessible residues	standard
10	Average area buried	standard
11	Alpha-helix (probabilities)	Chou & Fasman
12	Beta-sheet (probabilities)	Chou & Fasman
13	Beta-turn (probabilities)	Deleage & Roux
14	Total beta strand	standard
15	Antiparallel beta-strand	standard

Table 3.2: Secondary structure prediction tests

parsing the resulting data.

The machine learning algorithms did not perform very well on the ProtScale set. Moreover, obtaining the data was time-consuming. Consequently, we decided to try a third-party feature set that contained secondary structure information (from [12]). Given how well the machine learning algorithms did on the third-party set, and that it was a particularly difficult set (discussed below), we decided to use it as our baseline feature set. The results of the ProtScale set are given in Figure 4.1. It is evident from this venture that the choice of features is important, as is the choice of machine learning algorithm.

We decided to work with secondary structure data used by Ding and Dubchak for protein fold recognition, which are available for use [12]. This dataset was chosen because both the training and test sets are particularly difficult benchmarks. The training set is based on the PDB_select dataset developed by Hobohm et al. [26, 25]. No two proteins in this dataset have more than 35% sequence identity. The test set is a subset of the PDB-40D database chosen by the developers of the SCOP database [33], where proteins with more than 40% sequence similarity have been removed. For the rest of this thesis, we will refer to these sets together as the 40D-Select set.

Machine learning algorithms were trained on a subset of 40D-Select training set, and similarly tested the algorithms on a subset of the testing set. The properties that made the original datasets difficult (*i.e.*, low sequence identity) still hold for the subset used in our analysis. Since the domains were not ordered by architecture, each domain had to be manually annotated to include architecture information. A total of 118 domains were chosen from these sets as described below:

Of the 605 domains in the training file used in [12], 131 were manually annotated. Of those, 77 were in the Mainly Alpha class and fell into one of the architectures being studied. Of those, 1 domain had a PDB file that could not be used by FIRST and was thrown out of the analysis, leaving 76 instances from this file.

Of the 385 test instances, 104 were manually annotated and 50 fell in to one of the architectures being studied. Of these instances, 2 domains had PDB files that could not be used by FIRST and were thrown out of the analysis. That left 48 instances from this file. The total number of instances, then, is 126 domains.

The full list of domains used in the analysis are given below and separated by training and test set.

3sdha	1flp	2hbg	2mge	1eca	2gdm
1babb	1itha	1ash	1cpca	1grj1	1srya1
1ccr	1cxa	2pac	1enh	1lfb	1aplc
1hdp	1hcra	1ret	1msec1	1leb	1hsta
1hks	1erl	1erd	1lpe	1was	256ba
2hmza	2tmvp	3mdda1	1bcfa	1fha	1riba
1mmob	1bgeb	1lki	1huw	1gmfa	1rcb
1ilk	1rfba	1pou	1lmb3	2cro	1adr
4icb	1rtp1	1rec	2scpa	2sas	1mylb
1cmca	2gsta2	1gsra2	1gsq2	1lpt	1bip
1ezm2	8tlne2	7ccp	1lgaa	1mypa	2cp4
2hpda	1cpt	1poc	1poa	1ppa	1fc2d
1oxy1	1clc1	3hhr1	1ddt3		

Table 3.3: Domains in the training set (derived from the PDB_select set)

Python scripts were written to parse each annotated file (train and test) and store the pertinent information in comma-delimited form.

1hbg	1mba	1lh1	1baba	1alla	1dvh
5cytr	1cc5	351c	1gks	1aofa1	1etpa1
1yrna	1octc1	1res	1pdnc	1igna1	1bia_1
1lea	1aoy	1opc	1etd	1puee	2hts
1dpra1	1fow	2liga	1bbha	1cgo	1cpq
2hmqa	1vtmp	1buca1	1fapb	1bgc	1cnt1
1csga	2int	1hula	3inkc	1jli	1sra
1rro	1osa	1cpo_1	1vfba	1lla_2	1clc_2

Table 3.4: Domains in the test set (derived from the PDB-40D set)

3.4.2 Collecting the baseline feature set

The features chosen by Ding and Dubchack [12] formed our baseline feature set. These included the percentage composition of the 20 amino acids, transition frequencies, as well as polarity and hydrophobicity information. A more complete discussion of the feature vectors in these datasets can be found in [12].

3.4.3 Augmenting the baseline feature set with rigidity data

PDB files for structures that were resolved by X-ray crystallography do not contain information on hydrogen bonds. Therefore, we first ran the domains whose structures had been resolved by crystallography through the command line version of a program called REDUCE from Duke, which adds back hydrogen bonds using standard geometric techniques [51]. The latest version (August 2008) was used.

How a structure has been solved is noted in the PDB file, so in order to determine which structures had been solved by crystallography, we created a script that used regular expressions to find the appropriate annotation in the PDB file. This step is important because REDUCE could change the values for hydrogen bonds in files that already contain such information.

To automate this process, we wrote a bash script and used default values. On a set of 100 proteins, this step normally took a minute or less.

Rigidity data was collected using the command line version of FIRST from Flexweb [29, 14], using the default value of -1.0 kcal/mol as the energy cutoff for

every instance in our test and training sets.

A Perl script was used to create an XML file from the FIRST output containing body-bar-hinge information. A bash script was written to turn this into a batch process for all the proteins that were run through FIRST.

Each output file from the perl script was parsed with our own Python script to find the size of the largest rigid cluster. Specifically, we analysed the body information from this output and used Python's minidom module to read in the XML and find the size of the largest child node, which corresponded to the size of the largest rigid cluster (*i.e.*, the cluster with the most atoms in it).

Because the minidom module reads in the whole file, we had to make some modifications to the output file so that the script could complete in a reasonable amount of time. Since we only needed the body information, files larger than 10 MB were shortened to include only the body information. In order to do this truncation, we used Python's inputfile module to read the file in line by line. The pertinent information was then copied to a new file (so the original files were left intact). The size threshold of 10 MB was somewhat arbitrary. Through experimentation we discovered that it was about this point where the script started to hang up.

The rest of the rigidity data was taken from the results file outputted by FIRST. Python scripts were written to parse these files. In all, 9 features taken from the rigidity output were gathered by our scripts and are enumerated in Table 3.5.

N$^{o.}$	**Rigidity Information**
1	size of biggest rigid cluster
2	number of unique residues
3	number of isolated sites
4	number of isolated dimers
5	number of hydrogen bonds included
6	number of hydrophobic tethers included
7	total number of rigid clusters
8	number of stressed regions (regions with at least one overconstraint)
9	number of independent degrees of freedom

Table 3.5: Rigidity information gathered

Running FIRST alone on our protein set took about 10 or 15 minutes. Creating

the XML files itself was not a lengthy process, but the Perl script did have a tendency to make the computer overheat after running several analyses in succession, so we added a sleep command in the batch script for 3 seconds between each analysis. This meant that, for a protein set of 100, the process of creating the XML files took about 5–10 minutes.

With file truncation, finding the largest rigid cluster for each of 100 proteins took about 10–20 minutes, depending on the size of the original file. Parsing the other FIRST output and compiling that data into a new file took about 10 minutes in total.

3.4.4 Running the training and testing sets

The secondary structure and rigidity data were each stored in separate files, so scripts were written to compile this information into two files for both the training and test set. One file contained only the secondary structure data taken from the 40D-Select set, along with the architecture identifier. The other contained that data along with the rigidity data.

The machine learning software WEKA was used to perform the classification end [22]. The software had difficulty reading the CSV files used to store to feature sets, which is not an uncommon problem. For this reason, the CSV files were converted to WEKA's preferred ARFF format using a converter written by the WEKA team.

The training and test sets were manually inputted into WEKA for each of the machine learning algorithms listed in Section 2.4. The results of this analysis are given in Chapter 5. A screenshot of the output shown through the WEKA GUI for our augmented feature set on AdaBoost is given in Figure 3.3.

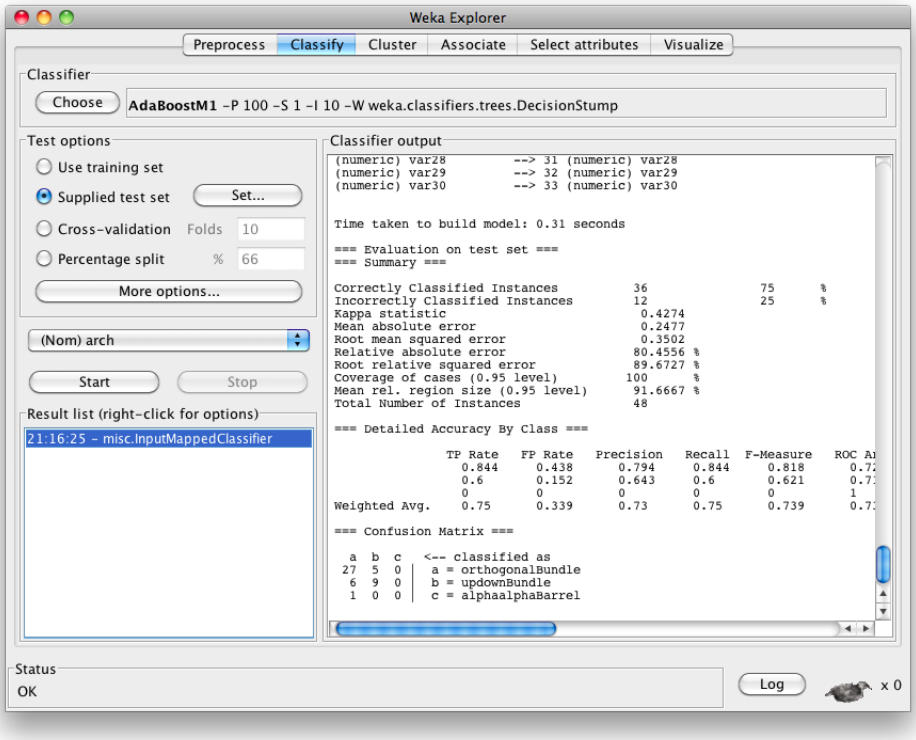


Figure 3.3: An example of WEKA's output from the GUI

Chapter 4

Results

Chapter 3 delineated the approach we took in this study, described the dataset used for the analysis, and discussed the tools we used. We now present the results at each step of the analysis in this chapter.

4.1 Comparison of baseline feature sets

As discussed in Chapter 3, we originally developed our own baseline feature set called the ProtScale set, and then compared the accuracies of the dataset on the 8 machine learning algorithms to those of the the 40D-Select set developed by [12] on the same machine learning algorithms.

We remind the reader that the latter was hand-chosen to be particularly difficult and that the domains in the ProtScale set are not the same as those in the 40D-Select set. Therefore, a direct comparison would be inappropriate. Nevertheless, we present the accuracies of both sets on each of the machine learning algorithms for general comparison in Table 4.1 below.

ML algorithm	ProtScale set	40D-Select set	% difference
Naïve Bayes	66%	58%	+8%
SVM (PolyKernel)	71%	67%	+4%
DecisionTree	67%	67%	0%
Attribute Selected Classifier	57%	67%	-10%
Multiclass Classifier	52%	58%	-6%
Multilayer Perceptron	71%	67%	+4%
Bagging	67%	67%	0%
AdaBoost	71%	67%	+4%

Table 4.1: Results of baseline feature sets: the ProtScale set and the 40D-Select set

As shown in Table 4.1, the ProtScale set did worse, the same, or only slightly better than the 40D-Select set, even though the latter was substantially more difficult. Given that there was no substantial difference in the accuracies, and given that the difficulty of the latter, we decided to use the 40D-Select dataset as our baseline feature set instead of the ProtScale set.

4.2 Results from FIRST

Rigidity information was gathered from FIRST, the rigidity analysis tool. Unfortunately, the software was not able to process all of the protein structure files. Table 4.2 summarises those proteins for which FIRST was not able to produce output and the reason FIRST gave for why this was so.

PDB ID	Error raised	Set
2lhb	The bond distance between intraresidue atoms 246 and 249 exceeds 6 Angstroms.	training
1mdb	The bond distance between intraresidue atoms 1209 and 1214 exceeds 6 Angstroms.	test
1fjl	Nitrogen 1542 has 5 or more covalently bonded neighbors.	test

Table 4.2: PDB files on which FIRST failed along with error messages (training and test sets)

For each of the PDB files that FIRST was able to process, 9 files were outputted. We were primarily interested in parsing the main results file. Example contents of this file for the PDB ID 1oxy are given in Figure 4.1. The full file is much longer than

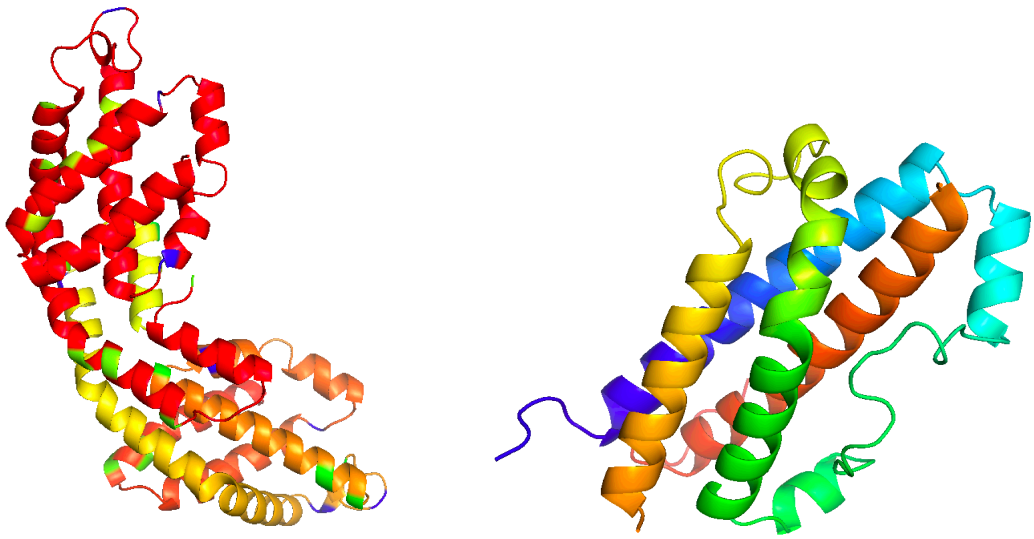
what can be seen in the example, but most of the information of interest appears near the top of the file, which can be seen in the figure.

```
# ------------------------------------------------------------------
# FILE INFO:

 Input file name: new-alphas/pdbfiles_merged/1oxy/1oxy.pdb
 Date: Mon Apr 11 16:23:31 2011

# ------------------------------------------------------------------
# INPUT PARAMETERS:

 Command line: "./FIRST-6.2.1-bin-32-gcc3.4.3-none/FIRST -L /home/cschirf/Documen
 Energy cutoff: -1.00 kcal/mol

# ------------------------------------------------------------------
# RESULTS:

           5006 Sites included in the calculation.
              0 Sites excluded in the calculation.
            909 Unique residues found.
            333 Isolated sites.
              0 Isolated dimers.
              0 Hydrogen bonds included.
            264 Hydrophobic tethers included.
              6 Stacked ring interactions included.
           2635 Rigid clusters.
           1037 Rigid clusters larger than size 1.
            137 Stressed Regions.
           2750 Total independent degrees of freedom.

# ------------------------------------------------------------------
# WARNINGS: LEVEL 1 warnings are most severe.

 LEVEL 1 WARNINGS: 0

 LEVEL 2 WARNINGS: 336
* Some atom names are not found in the PDB standard/het dictionary.  The
dicitonaries contain bonding connectivity for all residues and het groups
in the PDB.  Possible reasons why your atom name is not found are:
  i) You used what-if to add hydrogens (it changes atom names)
  ii) You have renamed your atoms in some other program
  iii) You have a hand-made het group or residue that is not from the PDB
Be aware that INTRA-residue bonding involving these atoms are distance-based,
and bonds are assumed to be rotatable.
Also, when FIRST looks for INTER-residue bonds, it looks for particular
```

Figure 4.1: Example of output in FIRST's main results file

FIRST also outputs a modified PDB file that, when displayed with PyMol (a protein visualisation software), allows users to visually inspect the results of the rigidity analysis. When viewed, different coloured region represent different rigid components found by the program. Example visualisations of some sample proteins are given in Figures 4.2a and 4.2b below.

The rigidity analysis for 1all almost partitions the domain into two equal halves, while the rigidity results for PDB ID 1huw show that almost every α-helix was found to be its own rigid structure.



(a) 1all (b) 1huw

Figure 4.2: Example of FIRST output visualised through PyMol, where different colours represent different rigid components

4.3 Comparison of baseline feature set and augmented feature set

After running both the training and test sets for our baseline feature set (baseFS) and our augmented feature set (augFS) on the machine learning (ML) algorithms detailed in Section 2.4, we obtained the following results:

ML algorithm	**Accuracy on baseFS (%)**	**Accuracy on augFS (%)**	**% diff.**
Naïve Bayes	58%	67%	+9%
SVM (PolyKernel)	67%	69%	+2%
DecisionTree	67%	67%	0%
Attribute Selected Classifier	67%	73%	+6%
Multiclass Classifier	58%	69%	+11%
Multilayer Perceptron	67%	67%	0%
Bagging	67%	67%	0%
AdaBoost	67%	75%	+8%

Table 4.3: Results of architecture level classification on baseline and augmented feature sets

As seen in Table 4.3, at worst, the added rigidity information did not change performance, and, at best, resulted in as much as a 11% improvement in accuracy. In the case of AdaBoost, we were able to reach a total accuracy of 75%.

The average improvement across the machine learning algorithms tested is 4.5%.

Chapter 5

Conclusions and Future Work

In this thesis, we looked at the three most populous **architectures** within the Mainly Alpha **class** of CATH: orthogonal bundle, up-down bundle, and alpha/alpha barrel. We developed training and test sets for a list of domains that fell within these domains. We had two training and two tests sets—one which corresponded to the baseline feature set and one which corresponded to the augmented feature set.

We trained 8 machine learning algorithms (Naïve Bayes, SVM, AdaBoost, bagging, an attribute selected classifier, a multiclass classifier, a decision table, and a multilayer perceptron) first on the baseline training feature set. We then tested this model on our baseline testing feature set to obtain baseline accuracy scores for the purposes of comparison. We performed the same process on our augmented feature sets and obtained those accuracy scores.

Our results show that augmenting current standard protein quantisations with rigidity information for whole proteins can improve classification accuracy, even on a particularly difficult protein set. Since we included rigidity information for whole proteins, this implies that the context in which a domain is found is a valuable source of information with regard to classification.

Given the range of improvement across the eight machine learning algorithms tested (0–11%), it is evident that the choice of algorithm is significant.

Future Work: This project was intended as an exploratory study of the applications of rigidity theory for protein classification. Consequently, the dataset we tested is relatively small compared to the total number of domains available in CATH alone. Furthermore, the rigidity information gathered is fairly basic.

Therefore, two obvious areas for further work are: to test our system on a larger domain collection (ideally, on all the domains currently in CATH) and to gather more sophisticated rigidity data. The latter may require some experimenting with the energy cutoff when running the proteins through FIRST. One idea for further tests is to try getting the largest n rigid clusters, instead of the single largest.

Another possible area of investigation would be to test the effect of rigidity data of a single domain instead of the whole protein. This would, again, entail some experimenting with the energy cutoff, as well as experimenting with running single chains through FIRST. It may turn out that a handful of domains would have to be analysed individually in order to determine the best value for the energy cutoff and to determine how the chains should be inputted (*e.g.*, if multiple chains should be aggregated into a single chain that represents a domain, or if chains should be analysed separately and the values for each chain then compiled into a single set of values). Ideally, work in this area should focus on choosing information in such a way that the requisite processes could be converted to an automated technique.

As we became more familiar with the domains in the architectures we studied, we noticed that many domains in the same architecture had shape characteristics distinctive to that architecture. It is then possible that classification accuracy could be improved with the addition of some kind of shape information. For example, if we could find an axis that characterises a rigid component, *e.g.*, an α-helix, we could compute the degree between the axes of several components. This could form a subset of features in our feature set. It would also be possible to try obtaining convex hull information for the overall shape of the domain.

It may also be worthwhile to see if adding rigidity information could improve the accuracy of current automation at the **class** level of the CATH hierarchy.

Appendix A

Selected Python Scripts

Python scripts were written to gather the necessary PDB and FASTA files and to automate the collection of data for the feature set.

Much of this data was collected from online tools, which required a programmatic method for filling out and submitting online forms. The library *mechanize* was used to simplify this process. It is necessary to install this library before running many of the scripts that gather feature data. It may be downloaded from `http://wwwsearch.sourceforge.net/mechanize/`.

A.1 Getting Files from the PDB Server

The following script gets PDB and FASTA files from the PDB server. The IDs of the proteins to download should be in a text file called `allproteins.txt` (or you may choose to rename it). The file should use the CATH's identification system for the protein IDs, as the first four characters of this ID correspond to the protein's PDB ID.

To increase efficiency, the script will only download the necessary files if they are not already present in the appropriate folder.

```
#####################################################################
#       PDB UTILITY
#
# Fetches and downloads PDB and FASTA files from the PDB server.
#
# Will only download those files if they have not already been
#  downloaded.
#
#
# Author: Courtney Schirf
# Last Revision: 7 December 2010
#####################################################################


import urllib2 #to open up URLs (current as of Python 3.0)
import os.path #to tell if a file already exists


PROT_FP = 'newAlphaProteins.txt'

#####################################################################
### METHOD DEFINITIONS


##
# Get the PDB (structural) files from the PDB server for every
# protein domain in our file
##
def getPDBFiles():
  #read in a file that has the names of all the protein domains
  # this file is has the CATH names
   proteinsfile = open(PROT_FP, 'r')
   pdbissuefile = open('pdbFilesErrors.txt', 'a')

   for line in proteinsfile:
    #the first 4 characters of the CATH name are the domain's PDB ID
    pdbnum = line[0:4]

    #try to connect to the pdb server
    # and get the file for the current PDB ID
    try:
```

```
    #this is the url for the pdb file on the pdb server
    #this code is just for .pdb files, but we
    # can get other kinds of files from the PDB server
    #e.g. to get the FASTA code (sequence data)
    #http://www.pdb.org/pdb/files/fasta.txt?structureIdList=pdbnum
    #the PDB ID is still the same :)
    pdburl = 'http://www.pdb.org/pdb/files/' + pdbnum + '.pdb'

    #copy pdb file from server to our own disk,
    # but only if we don't already have it
    filename = 'pdbfiles/' + pdbnum + '.pdb'

    if os.path.exists(filename):
     print 'We already have ' + pdbnum + '.pdb ' \
      + 'so we won\'t get it again'
    else:
     print "Getting " + pdbnum
     file = urllib2.urlopen(pdburl)
     fileout = open(filename, 'w')
     fileout.write(file.read())

     #close the file
     file.close()

   #if we can't connect to the server (may mean the url is wrong or
   # that the file does not exist) then catch the exception
   # and print a little message.
   except IOError:
    print 'That file does not exist.\n'
    pdbissuefile.write(pdbnum),

##
# Get the FASTA (sequence) files from the PDB server for every
# protein domain in our file
##
def getFASTAFiles():
  #read in a file that has the names of all the protein domains
  # this file is has the CATH names
  proteinsfile = open(PROT_FP, 'r')
  fastaissuefile = open('fastaFilesErrors.txt', 'a')
```

```
  for line in proteinsfile:
   #the first 4 characters of the CATH name are the domain's PDB ID
   pdbnum = line[0:4]
                                                                        86
   #try to connect to the pdb server
   # and get the file for the current PDB ID
   try:
    #Let's get the FASTA file, but only if we don't already have it
    fastaurl = 'http://www.pdb.org/pdb/files/fasta.txt?' \              91
     + 'structureIdList=' + pdbnum
    filename2 = 'fastafiles/' + pdbnum + '.fasta'

    if os.path.exists(filename2):
     print 'We already have ' + pdbnum + '.fasta so we won\'t ' \       96
      +'get it again'
    else:
     print "Getting FASTA file for " + pdbnum
     file = urllib2.urlopen(fastaurl)
     fileout = open(filename2, 'w')                                    101
     fileout.write(file.read())
     file.close()

   #if we can't connect to the server (may mean the url is wrong or
   # that the file does not exist) then catch the exception            106
   # and print a little message.
   except IOError:
    print 'That file does not exist.\n'
    fastaissuefile.write(pdbnum),
                                                                       111


##
# Check each PDB file and print out a message if the structure
# was found through X-ray crystallography (no hydrogen atoms)
##                                                                     116
def isPDBCrysto():
  #read in a file that has the names of all the protein domains
  # this file is has the CATH names
  proteinsfile = open(PROT_FP, 'r')
                                                                       121
  #create a file to hold the IDs of the proteins that are
  # made using X-ray crystallography (and thus require
```

```
  # hydrogens to be added)
  xrayfile = open('xrayproteins.txt', 'a')

  # TEMP counter
  count = 0

  for line in proteinsfile:
   pdbnum = line[0:4]
   filename = 'pdbfiles/' + pdbnum + '.pdb'

   if os.path.exists(filename):
    file = open(filename, 'r')
    content = file.read()
    isCrysto = (content.find('X-RAY')) != -1


    if isCrysto:
     print 'The pdb file ' + pdbnum + ' was made by X-ray ' \
      + 'crystallography'
     xrayfile.write(pdbnum + '\n')
     count += 1
     #print count

    file.close()

  print 'There were ' + str(count) + ' matches.'

### END METHOD DEFINITIONS
#########################################################################


class PDBGetter:

 def main():
  print str.upper('Fetching PDB files')
  getPDBFiles()
  print '\n'
  print str.upper('Fetching FASTA files')
  #print '\n'
  getFASTAFiles()
  print '\n'
```

```
 print str.upper('Checking if made by crystallography')
 #print '\n'
 isPDBCrysto()

if __name__ == '__main__':
    main()
```

A.2 Finding all Chains for a Given Domain

The following script uses the FASTA files to find all the chain letters and/or numbers for a given domain. It writes the domain's PDB number and list of chains to a new file.

This is useful for later calculations involving the domain as a whole, instead of by chain.

```
## ChainFinder
## output file format: pdbnum,ChainLetter,ChainLetter,ChainLetter,...

import fileinput
import os.path
import re


FASTA_DIR = "fastafiles/"
OUT_FP = "domainchains.txt"
IN_FP = "../all-alpha-proteins.txt" #"newAlphaProteins.txt"


def parseFastaFile(filepath, pdbnum):
 print filepath

 if not os.path.exists(filepath):
  return None

 out = pdbnum + ","

 inputobj = fileinput.input([filepath])
 regex = ">\w*"
 reggie = re.compile(regex)
```

```
                                                                    25
 for line in inputobj:
  a = reggie.search(line)

  if a is not None:
   out += line[6] + ","                                             30
 inputobj.close()

 out = out[0:len(out)-1]
 print out
 return out                                                         35


class ChainFinder:
 def main():
  allproteins = open(IN_FP, "r")                                    40
  #chainsfile = open(OUT_FP, "a")
  chainsfile = open(OUT_FP, "w")

  for line in allproteins:
   success = parseFastaFile(FASTA_DIR + line[0:4] + ".fasta", line[0:4])   45
   if success is None:
    print "Something went wrong."
   else:
    chainsfile.write(success)
    chainsfile.write("\n")                                          50
   print "\n"

  allproteins.close()
  chainsfile.close()
                                                                    55
 if __name__ == '__main__':
     main()
```

A.3 Finding the Size of the Largest Rigid Cluster

The following script uses the result of the `createXML.pl` script to find the size of the largest rigid cluster. It uses the minidom module to parse the XML file. However, this requires reading the whole file into memory, which slows down the script if

the files are large. So, if the file is greater than 20 MBs, `FirstXMLParser.py` will essentially create a new file that only copies over the essential XML tags.

```
import fileinput
import os
import re
from time import sleep
import types
from xml.dom import minidom


PROBLEMS_FP = "xmlParserProblemDomains4.txt"
STATS_FP = "xmlParserStats4.txt"


def findBiggestCluster():
 #file = open("../pythoncode/allproteins.txt", "r")
 file = open("../all-alpha-proteins.txt", "r")

 #Create both files now, because we'll be doing an append later.
 problemsfile = open(PROBLEMS_FP, "w")
 problemsfile.close()
 statsfile = open(STATS_FP, "w")
 statsfile.close()

 for line in file:
  pdbnum = line[0:4]
  #filepath = "../pdb_merged_FIRST_output/" + pdbnum + "/" + pdbnum + "
      _postPG_BBH.xml"
  filepath = "../pdb_alphas_first_output/" + pdbnum + "/" + pdbnum + "
      _postPG_BBH.xml"

  try:
   print filepath

   size = os.path.getsize(filepath)
   print "file size: " + str(size)
   mbs = (size/(1024*1024.0))
   print "File = %0.1f MB" % (mbs)

   if mbs > 15:
```

```
    print "The file for " + pdbnum + " is greater than 10 MBs."
    print "Preparing to create shortened file."                    38

    newfilepath = filepath[0:len(filepath)-len(".xml")] + "_mod.xml"
    print newfilepath

    newfile = open(newfilepath, "w")                               43
    regex = "<points>"
    reggie = re.compile(regex)
    inputobj = fileinput.input([filepath])

    for line in inputobj:                                          48
     a = reggie.search(line)

     if a is None:
      newfile.write(line)
     else:                                                         53
      newfile.write("\t</BBH>")
      break

    inputobj.close()
    filepath = newfilepath                                         58

#Parse the XML document
xmldoc = minidom.parse(filepath)

#Get all a list of all "pointSet" elements                         63
pointSetList = xmldoc.getElementsByTagName('pointSet')

#Variable to hold the size of the current biggest cluster
bigCluster = 0;
                                                                   68
for ps in pointSetList:
 numChildNodes = len(ps.childNodes)

 if numChildNodes > bigCluster:
  bigCluster = numChildNodes                                       73

print "The biggest cluster was: " + str(bigCluster) + "\n"

if bigCluster == 0:
```

```
    problemsfile = open(PROBLEMS_FP, "a")                                    78
    problemsfile.write(pdbnum + "\n")
    problemsfile.close()
   else:
    statsfile = open(STATS_FP, "a")
    statsfile.write(pdbnum + "," + str(bigCluster) + "\n")                   83
    statsfile.close()

  except IOError as error:
   pass

                                                                             88
  sleep(2)
 file.close()



class FirstXMLParser:                                                        93
 def main():
  findBiggestCluster()

 if __name__ == '__main__':
  main()                                                                     98
```

Appendix B

Selected Bash Scripts

B.1 Batch Processing using REDUCE (Hydrogen Bond Additions)

The following script runs all proteins whose structures were resolved using X-ray crystallography (and consequently do no have hydrogen bond data) through Kinemage's REDUCE software.

```
#!/bin/bash

while read line
 do
  # Can't have spaces around the equal sign. Otherwise, it will
  # think it's a command and try to run a program.
  #filename="../pdbfiles_reduced/${line:0:4}.pdb";
  filename="../pdb_alphas/${line:0:4}.pdb";
  echo "Running REDUCE on $filename";
  ./reduce.3.03.070307.macosx.i386 $filename | tee "../reduce_output/${line
      :0:4}_reduceOutput.txt"
 done < "../alpha-xrayproteins.txt"
 #done < "../pythoncode/xrayproteins.txt"
```

B.2 Batch Processing for FIRST Software

The following script runs all protein domains through FIRST using -1.0 kcal/mol as the energy cutoff. All output options are turned on (e.g., list of covalent bonds, a list of hydrophobic tethers). To allow batch processing, all interactions are turned off.

```
#!/bin/bash

#Create a directory for the output files if one does not already exist
# The follwoing check will NOT work for aliases / shortcuts / symbolic links
outdir="first_debug_output"
if [ ! -d "$outdir" ]; then
    mkdir "$outdir"
fi

#The name of the directory where the PDB files are stored.
directory="pdb_merged";

#The name of the file containing the PDB IDs of the domains to be run
pdbfile="../pythoncode/allproteins.txt";

while read line
 do
  pdbnum="${line:0:4}";
  filename="$directory/$pdbnum.pdb";

  #Create directories for each of the PDB files.
  # Must do this because some of the non-standard output files from
  # FIRST (e.g., hbonds.out) have the same name despite what PDB file
  # is given as input, so those files will constantly be over-written
  pdbdir="$pdbnum";

  if [ ! -d "$pdbnum" ]; then
   echo "creating a new directory named $pdbnum"

   cd "$directory"
   mkdir "$pdbnum"
   cd ".."
```

```
    #Move the PDB file into the new folder which bears its name
    mv "$filename" "$directory/$pdbnum/$pdbnum.pdb"
   else
    #If the directory already exists, just print a message saying so
    echo "The directory $pdbnum already exists"                    38
   fi

   #Re-direct the path to reflect the new changes
   filename="$directory/$pdbnum/$pdbnum.pdb"
                                                                   43
   #Run FIRST with all of the output options on and all interactions off
   echo "Running FIRST on $pdbnum";
   ./FIRST-6.2.1-bin-32-gcc3.4.3-none/FIRST -L /home/cschirf/Documents/thesis/
       first/FIRST-6.2.1-bin-32-gcc3.4.3-none "$filename" -E -1.0 -hbout -phout
        -covout -non | tee "first_output/$pdbnum-first-out.txt"
done < "$pdbfile"
```

Appendix C

Tutorial on Using the Scripts

C.1 Overview

The following sections explain in more detail the scripts used to collect and run the analysis described in Chapter 3 and describe how to run the analysis on a different set of proteins.

An executable bash script (`runall.sh`), contains all the right commands in the right order, but it is possible (and, indeed, likely) that something may go amiss during some step which will prevent subsequent steps from executing correctly or at all. We recommend that you try running this script on a short set of proteins to get a feel for how it works and where there may be problems with your own dataset before running it on a list of 100+ proteins.

Output on the command line should help you figure out which script failed (possible where it failed, as well). If this happens, we recommend you run that script and the subsequent ones individually. Table C.1 contains a summary of each step and scripts and files necessary to complete it. Furthermore, Table C.2 summarises for each step, which files are required for input and which should be expected as output.

Additionally, as of May 2011, FIRST does not run on the Mac platform, so either the scripts should be run on a Linux-box or the FIRST analysis should be

done separately (on a Linux platform) from the rest of the analayses. Naturally, lines in the `runall.sh` file can be commented out to achieve partial automation.

It must be noted that the scripts to compile and format this data have been less well written, and are heavily dependent on what software we used to perform the machine learning, as well as where we had gotten our secondary structure information from. So, you might need to write your own script to do this compilation and formatting.

However, the various files we used to compile this information have most of the necessary core functionality, and could easily be re-worked to fit a different application. These files are `AugRigidty.py`, `Domain.py` (a Python class), and `Controller.py`. The `AuxMethods.py` file should also prove helpful.

C.2 Preliminary steps

Before we can collect any secondary structure or rigidity data, we first need to gather all of our initial files. This includes creating a list of the proteins we want and then downloading the corresponding structure and amino acid composition files (PDB and FASTA, respectively).

C.2.1 Creating a list of the proteins

The most important file in this process is the `allproteins.txt` file, which contains a list of all the proteins on which the analyses are to be run. In this file, each protein number must be written on a new line. The protein number can either the CATH number or the PDB ID, because the first 4 characters of the CATH number is the PDB ID (and the script will only read the first 4 characters of the line). Additionally, there should be a blank line at the end of the file since several bash scripts read this file.

If you wish to change the name of the file or put this file in a directory different from that of the scripts (which we discourage), then you will have to change some

of the source code to reflect the change.

You'll probably also need to create another version of this file, but where each line will also contain the CATH architecture number for that protein (*e.g.*, 1.10) separated from the PDB ID/CATH number by white space; we used tabs. This file must be called `allproteins-annotated.txt`, and should be in the same directory as `allproteins.txt`. This information should be used to create the final data compilation.

Our allproteins file was created by hand.

C.2.2 Collecting the PDB and FASTA files

After the allproteins file has been made, you're ready to download the PDB and FASTA files. If you're running this from the `runall.sh` script, a directory named `output` will be created for you. If not, then you should create a directory with that name.

Then run the `pdbUtitlity.py` script. This script should be in the same directory as the allproteins file. It will create directories in the output folder called `pdbfiles` and `fastafiles`, and query the PDB server for both files and place them into the appropriate directories. If the script cannot, for whatever reason, find or obtain either a PDB or FASTA file, it will log that information in `pdbFileErrors.txt` and `fastaFileErrors.txt`, respectively.

This script will also look at the notes sections in each PDB files to determine if the protein's structure was resolved using X-ray crystallography. Recall from Section 3.4.3 that proteins for which this is true do not contain hydrogen bond information in their PDB files, and that this data will have to be added by REDUCE in a later step. Proteins that fall under this category are logged in a file called `xrayproteins.txt`, which is used later.

All of the log files are in the `output` directory.

C.2.3 Finding the chain names for each protein

If you are going to gather secondary structure information as well as rigidity information, you will need to also run the `ChainFinder.py` script, which parses the FASTA files to find out the names of each chain for each protein in the set and compiles that information into a file called `domainchains.txt`.

C.3 Collecting secondary structure information from ProtScale

If you would like to gather secondary structure information from ProtScale, run the `ProtScaleFormCompleter.py`. This script will create a folder called `prot_scale_output`. Within this directory, it will create a new folder or each PDB ID of the form `$pdbID_protscale`[1] just before it gathers information for that protein.

This script fills out the ProtScale form for each protein chain (the server can only analyse one chain at a time) and writes the output to a file within its `$pdbID_protscale` folder of the form `$pdbID$chainID_$scaleName`.

Allow lots of time for this script, as it takes at least several hours to run. Our first time on this step took a good 3 days to complete. You also might need to restart the script at a different point in the allproteins file, because sometimes the ExPASy server will be rebooted.

C.4 Collecting rigidity data

C.4.1 Prepping the data with REDUCE

Run the `reduce_batch.sh` script, which relies on the `xrayproteins.txt` file created by the `pdbUtility.py` script. Because REDUCE changes the actual PDB files, we suggest making a copy of the PDBs and run REDUCE on the copies. If you are

[1]The $ symbol will be used to refer to variable names.

running the `runall.sh` file, then this will be done for you. The new directory is called `pdbfiles_reduced` and is found in the `output` directory.

All of REDUCE's output is logged into separate files for each protein and saved to a new directory called `reduce_output`. If an error occurs at this step, these output files might contain information as to why.

Then you will need to merge the REDUCE'd PDBs with the non-REDUCE'd PDBs. This step is accomplished with the `mergeReducedAndRegPDBs.py`, which creates a new directory called `pdbfiles_merged`, and copies over the PDB files from the appropriate directories.

C.4.2 Run FIRST

Once we've REDUCE'd our protein files, we are ready to run FIRST. As mentioned above, this software does not work on the Mac platform, so you'll have to find a Linux machine for this step.

The magical script you'll need to run is `first_batch.sh`, which will create a new directory within the `pdbfiles_merged` folder for each domain. It will then copy the appropriate PDB file into each directory and then run the analysis on the PDB file. The reason for this is that the filenames for optional FIRST output are not distinguished by PDB ID, so when running a batch script, the same file will be overwritten multiple times.

C.4.3 Get the size of the largest rigid cluster for each protein

Run the `RunAllCreateXML.sh` file to create the body-bar-hinge (BBH) XML files for each protein in the set. This file requires the `createXML.pl` file, so make sure they're both in the same directory (or change the source code).

This script will create a file of the form `$pdbID_postPG_BBH.xml`, which is found in the individual PDB folders.

Then parse those files with the `FirstXMLParser.py` file. If the original XML file is over 10 MBs in size, the truncated version of the file will be saved in the form

`$pdbID_postPG_BBH_mod.xml`. The original file is left untouched.

The parser script will create a new directory called `xml_work` in the `output` directory. Within that directory, it creates two new files—`xmlParserStats.txt` and `xmlParserProblemDomains.txt`. The first of these contains the largest rigid cluster data, where each line is of the form `$pdbID,$sizeOfLargestRigidCluster`. The second logs any files the script had trouble parsing.

C.4.4 Parse the rest of the FIRST output

Run the `RigidityDataCalculator.py` file to compile all the rigidity data for each protein into a single file called `RigidityData.txt`. The script requires the `xmlParserStats` file created in the previous step in order to get the size of the largest rigid cluster for each protein.

This file is comma-delimited, where the first element of each line is the PDB ID followed by each of the rigidity values. The order these values are in is noted in the comments section of the script.

Errors with calculating this data is logged in the `ridigityDataCalcProblems.txt` file. Outputted files are put in a new directory called `rigidty_output`.

C.4.5 Compiling the output into a single file

As mentioned above, this is a more application-specific process, so we'll leave it to the user to complete this last part.

C.5 Reference charts

C.5.1 Subprocesses reference

The following chart contains the basic subprocesses in order of how they should be run.

N$^{o.}$	Step	Description	Required files
1	Create the allproteins file	Create a file with the PDB ID or CATH number of every protein in your set	*none*
2	Collect the PDB & FASTA files		`allproteins.txt` `pdbUtility.py`
3	Run `ChainFinder`	Determine the chain labels for proteins	`allproteins.txt` `ChainFinder.py`
4	Run ProtScale script	Run and collect secondary structure information	`allproteins.txt` `ProtScaleFormCompleter.py`
5	Run REDUCE	Add back hydrogen bonds to proteins whose structures were resolved using X-ray crystallography	`allproteins.txt` `xrayproteins.txt`
6	Merge PDBs	Merge the non-REDUCE'd and REDUCE'd PDB files into a single location	`allproteins.txt` `mergeReducedAndRegPDBs.py`
7	Run FIRST	Collect rigidity data	`allproteins.txt` `first_batch.sh`

8	Create XML files from FIRST output		`allproteins.txt` `createXML.pl` `runAllCreateXML.sh`
9	Parse XML files	Get the largest rigid cluster from each protein by reading the XML file	`allproteins.txt` `FirstXMLParser.py`
10	Parse FIRST output and compile rigidity data		`allproteins.txt` `RigidityDataCalculator.py`

Table C.1: Reference chart for running this analysis on a given set of proteins

C.5.2 I/O reference

The following chart summarises the input and output (files and directories) for each step.

N$^{o.}$	Step	Input files	Output files
11	Create the all-proteins file	*none*	`allproteins.txt` `allproteins-annotated.txt`
12	Collect the PDB & FASTA files	`allproteins.txt` `pdbUtility.py`	`pdbfiles` `fastafiles` `xrayproteins.txt` `pdbFileErrors.txt` `fastaFileErrors.txt`
13	Run `ChainFinder`	`allproteins.txt` `ChainFinder.py`	`domainchains.txt`
14	Run ProtScale script	`allproteins.txt` `ProtScaleFormCompleter.py`	`prot_scale_output` (and appropriate files and subdirectories)
15	Run REDUCE	`allproteins.txt` `xrayproteins.txt`	*alters PDB files*
16	Merge PDBs	`allproteins.txt` `mergeReducedAndRegPDBs.py`	`pdbfiles_merged`
17	Run FIRST	`allproteins.txt` `first_batch.sh`	individual protein folders, appropriate files from FIRST output
18	Create XML files from FIRST output	`allproteins.txt` `createXML.pl` `runAllCreateXML.sh`	`_postPG_BBH.xml` files
19	Parse XML files	`allproteins.txt` `FirstXMLParser.py`	`xml_work` `xmlParserStats.txt` `xmlParserProblemDomains.txt`

20	Parse FIRST output and compile rigidity data	`allproteins.txt`
		`RigidityDataCalculator.py`
			`ridigity_output`
			`RigidityData.txt`
			`rigidtyDataCalcProblems.txt`

Table C.2: Reference chart for the input and output files and directories for each step

Bibliography

[1] Nickolai Alexandrov and Ilya Shindyalov. Pdp: Protein domain parser. *Bioinformatics*, 19(3):429–430, 2003.

[2] Erez Berkovich, Hillel Pratt, and Moshe Gur. Face recognition with biologically motivated boost features. In *Cognitive Vision*, 4th International Workshop, ICVW. Springer, May 2008.

[3] Leo Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

[4] Leo Breiman. Arcing classifiers. *The Annals of Statistics*, 26(3):801–849, 1998.

[5] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

[6] Neil A. Campbell and Jane B. Reece. *Biology*. Benjamin Cummings, 7th edition, 2004.

[7] I-Fang Chung, Chuen-Der Huang, Ya-Hsin Shen, and Chin-Teng Lin. Recognition of structure classification of protein folding by NN and SVM hierarchical learning architecture. In *ICANN/ICONIP'03 Proceedings of the 2003 joint international conference on Artificial neural networks and neural information processing*, pages 1159–1167. Springer-Verlag, 2003.

[8] Pietro Cozzini, Glen E. Kellogg, Francesca Spyrakis, Donald J. Abraham, Gabriele Costantino, Andrew Emerson, Francesca Fanelli, Holger Gohlke, Leslie A. Kuhn, Garrett M. Morris, Modesto Orozco, Thelma A. Pertinhez,

Menico Rizzi, and Christoph A. Sotriffer. Target flexibility: An emerging consideration in drug discovery and design. *Journal of Medical Chemistry*, 51(20):6237–6255, October 2008.

[9] Gordon M. Crippen. The tree structural organization of proteins. *J. Mol. Bio.*, 126(3):315–332, December 1978.

[10] Alison L. Cuff, I. Sillitoe, Tony Lewis, Andrew B. Clegg, Robert Rentzsch, Nicholas Furnham, Marialuisa Pellegrini-Calace, David Jones, J. M. Thornton, and C. A. Orengo. Extending CATH: increasing coverage of the protein structure universe and linking structure with function. *Nucleic Acids Research*, 39:420–426, November 2010.

[11] Alison L. Cuff, I. Sillitoe, Tony Lewis, Oliver C. Redfern, Richard Garratt, J. M. Thornton, and C. A. Orengo. The CATH classification revisited: architectures reviewed and new ways to characterize structural divergence in superfamilies. *Nucleic Acids Research*, 37:310–314, 2009.

[12] Chris Ding and Inna Dubchak. Multi-class protein fold recognition using support vector machines and neural networks. *Bioinformatics*, 17(4):349–358, 2001.

[13] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley, 2001.

[14] Michael F. Thorpe et al. FIRST: Floppy Inclusions and Rigid Substructure Topography. http://flexweb.asu.edu/software/first/.

[15] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Thirteenth International Conference on Machine Learning*, pages 148–156, 1996.

[16] Yoav Freund and Robert E. Schapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780, September 1999.

[17] E. Gasteiger, C. Hoogland, A. Gattiker, S. Duvaud, M. R. Wilkins, R. D. Appel, and A. Bairoch. Protein identification and analysis tools on the expasy server.

In John M. Walker, editor, *The Proteomics Protocols Handbook*, pages 571–607. Humana Press, 2005.

[18] Holger Gohlke, Leslie A. Kuhn, and David A. Case. Change in protein flexibility upon complex formation: analysis of Ras-Raf using molecular dynamics and a molecular framework approach. *Proteins*, 56:322–337, 2004.

[19] Jack Graver, Brigitte Servatius, and Herman Servatius. *Combinatorial Rigidity*, volume 2 of *Graduate Studies in Mathematics*. American Mathematical Society, 1993.

[20] Jenny Gu and Philip E. Bourne, editors. *Structural Bioinformatics*. John Wiley and Sons, Inc, Hoboken, New Jersey, second edition, 2009.

[21] Jayavardhana Gubbi, Alistair Shilton, and Marimuthu Palaniswami. Kernel methods in protein structure prediction. In Yan-Qing Zhang and Jagath C. Rajapakse, editors, *Machine Learning in Bioinformatics*, chapter 9. Wiley, 2009.

[22] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. Weka. In *The WEKA data mining software: an update*, volume 11 of *SIGKDD Explorations*, 2009.

[23] Oktie Hassanzadeh. Automated protein structure classification: A survey. University of Toronto, January 2008.

[24] B. M. Hespenheide, A. Rader, Michael F. Thorpe, and Leslie A. Kuhn. Identifying protein folding cores from the evolution of flexible regions during unfolding. *Journal of Molecular Graphics and Modelling*, 21:195–207, 2002.

[25] Uwe Hobohm and Chris Sander. Enlarged representative set of protein structures. *Protein Science*, 3:522–524, 1994.

[26] Uwe Hobohm, Michael Scharf, Reinhard Schneider, and Chris Sander. Selection of representative protein data sets. *Protein Science*, 1:409–417, 1992.

[27] Sujun Hua and Zhirong Sun. A novel method of protein secondary structure prediction with high segment overlap measure: support vector machine approach. *J. Mol. Bio.*, 308:397–407, 2001.

[28] Donald Jacobs, A. Rader, M. Thorpe, and Leslie Kuhn. Protein flexibilty predictions using graph theory. *Proteins*, 44:150–165, 2001.

[29] Donald J. Jacobs, Leslie A. Kuhn, and Michael F. Thorpe. Flexible and rigid regions in proteins. *Rigidity Theory and Applications*, pages 357–384, 1999.

[30] Sotiris B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica (Slovenia)*, 31(3):249–268, 2007.

[31] Gerard Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics*, 4:331–340, 1970.

[32] Audrey Lee and Ileana Streinu. Pebble game algorithms and sparse graphs. *Discrete Mathematics*, 308(8):1425–1437, 2008.

[33] Loredana Lo Conte, Bart Ailey, Tim J. P. Hubbard, Steven E. Brenner, Alexey G. Murzin, and Cyrus Chothia. SCOP: A structural classification of proteins database. *Nucleic Acids Research*, 28(1):257–259, 2000.

[34] A. D. Michie, C. A. Orengo, and J. M. Thornton. Analysis of domain structural class using an automated class assignment protocol. *J. Mol. Bio*, 262:168–185, 1996.

[35] Natasha Mohanty and Audrey Lee-St. John. Shape-based image classification. CS89 Final Project at UMass Amherst.

[36] Sreerama K. Murthy. Automatic construction of decision trees from data: A multi-discipline survey. *Data Mining and Knowledge discovery*, 2:345–389, 1998.

[37] C. St. J. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal London Math. Soc.*, 36:445–450, 1961. Characterization graphs containing k

edge-disjoint spanning trees with counting properties (including (k, k)-sparsity) and partition results).

[38] C. A. Orengo, Nigel P. Brown, and William R. Taylor. Fast structure alignment for protein databank searching. *Proteins*, 14:139–167, 1992. difficult paper to get ahold of. there is a scanned copy attached to this entry.

[39] C. A. Orengo, A. D. Michie, S. Jones, D. T. Jones, M. B. Swindells, and J. M. Thornton. CATH: A hierarchic classification of protein domain structures. *Structure*, 5:1093–1108, August 1997.

[40] S. K. Pal and S. Mitra. Multilayer perceptron, fuzzy sets, and classification. *IEEE Transactions on Neural Networks*, 3(5):683–697, September 1992.

[41] F. M. G. Pearl, C. F. Bennett, J. E. Bray, A. P. Harrison, N. Martin, A. Shepherd, I. Sillitoe, J. M. Thornton, and C. A. Orengo. The CATH database: an extended protein family resource for structural and functional genomics. *Nucleic Acids Research*, 31(1):452–455, 2003.

[42] Jane S. Richardson. The anatomy and taxonomy of protein structure. In *Advances in protein chemistry*, volume 34. Academic Press, Inc., 1981.

[43] Asim S. Siddiqui and Geoffrey J. Barton. Continuous and discontinuous domains: An algorithm for the automatic generation of reliable protein domain definitions. *Protein Science*, 4:872–884, 1995.

[44] Swiss Institute of Bioinformatics. ExPASy Proteomics Server, 2005.

[45] H. Takagi, H. Shiomi, H. Ueda, and H. Amano. A novel analgesic dipeptide from bovine brain is a possible met-enkephalin releaser. *Nature*, 282(5737):410–2, November 1979.

[46] Tiong-Seng Tay. Rigidity of multi-graphs. I. Linking rigid bodies in n-space. *Combinatorial Theory Series*, B(26):95–112, 1984.

[47] Tiong-Seng Tay and Walter Whiteley. Recent advances in the generic rigidity of structures. *Structural Topology*, 9:31–38, 1984.

[48] The Orengo Group. CATH. Website, 2008. `http://www.cathdb.info/`.

[49] William T. Tutte. On the problem of decomposing a graph into n connected factors. *Journal London Math. Soc.*, 142:221–230, 1961. Characterization of graphs that can be decomposed into k edge-disjoint spanning trees with (k, k)-sparsity.

[50] David Whitford. *Proteins: structure and function.* John Wiley and Sons, Ltd., The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, England, 2005.

[51] J. Michael Word, Simon C. Lovell, Jane S. Richardson, and David C. Richardson. Asparagine and glutamine: Using hydrogen atom contacts in the choice of side-chain amide orientation. *J. Mol. Biol.*, 285:1735–1747, 1999.

[52] Mohammed J. Zaki and Christopher Bystroff, editors. *Protein structure prediction.* Humana Press, 2nd edition, 2008.

[53] Nela Zavaljevski, Fred J. Stevens, and Jaques Reifman. Support vector machines with selective kernel scaling for protein classification and identification of key amino acid positions. *Bioinformatics*, 18(5):689–696, 2002.