Spring 2013

Application of Query-Based Qualitative Descriptors in Conjunction with Protein Sequence Homology for Prediction of Residue Solvent Accessibility

Reecha Nepal
San Jose State University

APPLICATION OF QUERY-BASED QUALITATIVE DESCRIPTORS IN CONJUNCTION WITH PROTEIN SEQUENCE HOMOLOGY FOR PREDICTION OF RESIDUE SOLVENT ACCESSIBILITY

A Thesis

Presented to

The Faculty of the Department of Chemistry

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Reecha Nepal

May 2013

The Designated Thesis Committee Approves the Thesis Titled

APPLICATION OF QUERY-BASED QUALITATIVE DESCRIPTORS IN CONJUNCTION WITH PROTEIN SEQUENCE HOMOLOGY FOR PREDICTION OF RESIDUE SOLVENT ACCESSIBILITY

by

Reecha Nepal

APPROVED FOR THE DEPARTMENT OF CHEMISTRY

SAN JOSE STATE UNIVERSITY

May 2013

Dr. Brooke Lustig	Department of Chemistry
Dr. Daryl K. Eggers	Department of Chemistry
Dr. Marc d'Alarcao	Department of Chemistry

ABSTRACT

APPLICATION OF QUERY-BASED QUALITATIVE DESCRIPTORS IN CONJUNCTION WITH PROTEIN SEQUENCE HOMOLOGY FOR PREDICTION OF RESIDUE SOLVENT ACCESSIBILITY

by Reecha Nepal

Characterization of relative solvent accessibility (RSA) plays a major role in classifying a given protein residue as being on the surface or buried. This information is useful for studying protein structure and protein-protein interactions, and it is usually the first approach applied in the prediction of 3-dimensional (3D) protein structures.

Various complicated and time-consuming methods, such as machine learning, have been applied in solvent-accessibility predictions. In this thesis, we presented a simple application of linear regression methods using various sequence homology values for each residue as well as query residue qualitative predictors corresponding to each of the 20 amino acids. Initially, a fit was generated by applying linear regression to training sets with a variety of sequence homology parameters, including various sequence entropies and residue qualitative predictors. Then the coefficients generated via the training sets were applied to the test set, and, subsequently, the predicted RSA values were extracted for the test set. The qualitative predictors describe the actual query residue type (*e.g.*, Gly) as opposed to the measures of sequence homology for the aligned subject residues. The prediction accuracies were calculated by comparing the predicted RSA values with NACCESS RSA (derived from X-ray crystallography). The utilization of qualitative predictors yielded significant prediction accuracy.

ACKNOWLEDGEMENT

First of all, I would like to express my gratitude towards my research advisor, Dr. Brooke Lustig, for guiding me through all of this work. I would like to thank him for his patience, the amount of time he spent with me, and all the opportunities he provided me. I will forever be thankful for his encouragement and his positive attitude when things were not working as expected. I would also like to thank my M. S. thesis committee members, Dr. Daryl Eggers and Dr. Marc d'Alarcao, first for agreeing to be in my committee. Second, for spending time in reading and providing valuable feedback on my thesis. I would also like to thank my parents and my brother for their unshakable believe in me. Finally, I would like to extend my heartfelt gratitude towards my husband, Sailesh Agrawal, without whose help, support and encouragement this work would not have been possible.

CONTENTS

List of Abbreviations ..... **viii**

List of Figures ..... **ix**

List of Tables ..... **xi**

1. **Introduction** **1**
   - **1.1 Protein Structure Prediction Methods** **1**
     - 1.1.1 Experimental Approaches 2
     - 1.1.2 First Approaches in Protein Structure Determination 3
   - **1.2 Sequence Entropy** **5**
   - **1.3 Overview** **7**
   - **1.4 Organization of the Thesis** **8**

2. **Methods** **9**
   - **2.1 Protein Sets and Preparations** **9**
   - **2.2 Residue Packing Density** **13**
   - **2.3 Sequence Variability** **14**
   - **2.4 Homology-Based Parameters** **15**
   - **2.5 RSA Calculations** **16**
   - **2.6 Determination of Qualitative Predictors** **17**
   - **2.7 PSI-BLAST Calculations** **19**
   - **2.8 Accuracy Calculations** **22**
   - **2.9 Aggregate Analysis and Correlation Plots** **23**
   - **2.10 Frequency Distributions** **23**
   - **2.11 Assimilation of Additional Methods to Improve Accuracy** **24**
     - 2.11.1 Incorporation of Tertiary Protein Structure Information 24
     - 2.11.2 Additional Models Applied 26
     - 2.11.3 Incorporation of a Categorized Protein Data Set 27

3. **Results** **29**
   - **3.1 Characterization of Protein Lists** **29**
   - **3.2 Accuracy of Results** **53**
   - **3.3 Outcome of Additional Methods to Improve Prediction Accuracy** **57**
     - 3.3.1 Outcome of Additional Models Applied 57
     - 3.3.2 Outcome of Categorized Protein Data Set 58
     - 3.3.3 Use of All 618 PDB IDs as Training and 215 as Test 60

3.4 Incorporation of the Categories to the Existing Data Sets 62
4. Discussion 72
4.1 Prediction of RSA 72
5. Future Studies 79
6. Conclusions 80
Bibliography 82
Appendices 86
A. Program Listings 86
B. 1363 PDB Table 136

List of Abbreviations

Abbreviations	Full Form
PDB	Protein Data Bank
FSR	Fraction Small Residues
FSHP	Fraction Strongly Hydrophobic
NSHP	Non-Strongly Hydrophobic
FA	Fraction Alanine
FG	Fraction Glycine
E6	6-point Sequence Entropy
E20	20 term Sequence Entropy
RSA	Relative Solvent Accessibility
PB	Protein Binding

List of Figures

Figure 2.1 Flowchart of steps involved in the generation of the 1363 training data set based on the list of proteins from Bondugula et al. (2011). 12

Figure 2.2. Sample regression fit for 73,734 query residues from the 268 training data set. 20

Figure 2.3: Sample regression fit for 319,551 query residues from the 1363 training data set. 21

Figure 3.1. Frequency distributions for the characterization of the 268 learning set list of proteins. 31

Figure 3.2. Frequency distributions for the characterization of the 1363 learning set list of proteins. 33

Figure 3.3. Frequency distributions of entropies and homology-based parameters of the 268 training set list of proteins. 36

Figure 3.4. Frequency distributions of entropies and homology-based parameters of the 268 data set for the two major regions, Region I and Region II. 38

Figure 3.5. Frequency distributions for entropies and homology-based parameters of the 1363 learning set list of proteins. 40

Figure 3.6. Frequency distributions of entropies and homology-based parameters of the 1363 training data set for the two major regions, Region I and Region II. 41

Figure 3.7. Frequency distribution of NACCESS RSA values for various RSA ranges for the 268 training data set. 42

Figure 3.8. Frequency distribution of RSA values for various RSA ranges for the 1363 training data set. 43

Figure 3.9. Frequency distribution comparison of NACCESS RSA values and predicted RSA values for the 215 test set using the 268 training set. 44

Figure 3.10. Frequency distribution comparison of NACCESS RSA values and predicted RSA values for the 215 test set using the 1363 training set. 46

Figure 3.11. Comparison of combined aggregate correlation plots of sequence entropy and other homology-based parameters for the 268 training set. 48

Figure 3.12. Combined aggregate correlation plots of sequence entropy and other homology-based parameters for the 1363 training set. 49

Figure 3.13. Density—relative surface accessibility comparison for the 268 training set. 51

Figure 3.14. Density—relative surface accessibility comparison for the 1363 training set. 52

List of Tables

Table 3.1: Comparison of regression accuracy using manually generated BLAST output calculation and automatically generated BLAST output calculations. 55

Table 3.2: Summary of regression accuracy for the 12 models tested. 56

Table 3.3: Regression accuracy table for additional models applied. 58

Table 3.4: Protein binding category PDB ID matches between the 618 Hotpatch protein PDB IDs (Petit et al., 2007) and the 268 and 1363 training sets and 215 test set. 59

Table 3.5: Regression accuracy table for protein binding model. 60

Table 3.6: Description of 618 PDB IDs as training and 215 PDB IDs as test regression analysis calculations. 61

Table 3.7: Regression accuracy table of PDB IDs of the four groups: 1. generic 2. protein binding 3. oligomers 4. and generic without PB and Oligomers used in the regression analysis. 62

Table 3.8: PDB ID matches between the 618 data set with three data sets (215, 268, and 1363) for the protein binding and oligomer categories. 65

Table 3.9: PDB ID matches between the 618 data set with three data sets (215, 268, 1363) between all categories. 66

Table 3.10: PDB ID matches between the 618 data set and the three data sets (215, 268, and 1363) between all categories without the protein binding and oligomer matches. 67

Table 3.11: Regression accuracy for generic category using common PDB IDs between the 618 data set and the 215, 268, and 1363 data sets for the two regression models. 68

Table 3.12: Regression accuracy for the protein binding category using common PDB IDs between the 618 data set and the 215, 268, and 1363 data sets for the two regression models. 69

Table 3.13: Regression accuracy for the oligomer category using common PDB IDs between the 618 data sets and the 215, 268, and 1363 data sets for the two regression models. 70

Table 3.14: Regression accuracy for generic category excluding protein binding and oligomer categories using common PDB IDs between the 618 data sets with 215, 268 and 1363 data sets for the two regression models. 71

1. Introduction

Proteins, polymers of 20 amino acids, are biological macromolecules essential to life that have a variety of functions within each cell. Proteins serve both mechanical (actin in muscle that aids in mobility) and structural (involved in system of scaffolding in cell to maintain shape) functions. Many proteins are enzymes that are involved in vital biochemical processes such as metabolism. In addition to these roles, proteins are also involved in cell signaling, immune responses, cell adhesion, and cell cycle.

1.1 Protein Structure Prediction Methods

Protein structure is directly related to protein function. The study of protein structure and function could provide much-needed insight in the role of proteins *in vivo*. Knowledge of protein structure is fundamental for uncovering mechanisms of actions for various protein functions, exploring protein-protein interaction, predicting protein hydration sites, and characterizing hydrophobic clusters in proteins. Furthermore, structure explorations could facilitate the discovery and development of various drugs, so as to aid in the discovery of solutions to different protein malfunction and protein absence disorders in humans. Protein structure and functions are studied both experimentally and computationally.

1.1.1 Experimental Approaches

Protein structure determinations have usually involved expensive and time-consuming methods such as X-ray diffraction analysis, immunohistochemistry, site-directed mutagenesis, chromatography, and nuclear magnetic resonance (NMR). Processes such as these provide high-quality 3D protein structure determinations with precise accuracy and sensitivity. However these processes can be extremely expensive and time consuming. As a result, the corresponding 3D structures of the majority of proteins have not yet been characterized. In addition, because there can be significant difficulty in characterizing protein structures via X-ray, there is a desire for protein prediction using sequence and/or sequence homology (Dale et al., 2003).

Historically, a vast majority of proteins in the Protein Data Bank (PDB) are determined via X-ray (82%) with NMR being utilized for most of the remaining structures (Berman et al., 2000). The success rate of a high-resolution 3D protein structure analysis is very low, with only 2–10% of protein targets resulting in high-resolution protein structures (Mizianty and Kurgan, 2011). In order to deduce the correct 3D structure of a protein, it is essential to have a high-resolution crystal structure of the protein. Protein crystal formation is an active field of biological science, and much work is required before the low yield of protein crystallization can be improved. Moreover, there are certain groups of proteins, such as some membrane ribosomal proteins, for which crystal structure deduction is problematic (Gluehmann et al., 2001; Mizianty and Kurgan, 2011). Often these difficulties result from the production of diffraction-quality crystals (Chayen and Saridakis, 2008). The target protein crystallization process can be

expensive and time consuming because it involves trial and error until the best-quality crystal is obtained.

Interestingly, the use of computational methods in protein crystallization determination can aid in directing the most efficient use of resources in the current coordinated effort to determine high-resolution 3D structures for the whole proteome. One of the primary examples of this is the establishment of the Protein Structure Initiative (PSI), whose main goal is to determine 3D structures of proteins on a large scale (Berman, 2008). This has also greatly improved success rates of structural methods (Graebsch et al., 2010; Savchenko et al., 2003). One offshoot of X-ray structure determination involves a special class of computational modeling that is sometimes referred to as protein homology modeling.

Here, homology involves aligning 2 or more sequences of a known X-ray structure, essentially "stitching" them together, and subsequently optimizing the resulting form. One such homology-modeling program is called MODELLER (Eswar et al., 2006). It implements a process similar to that of 2D NMR called spatial restraints. In this process, a set of geometrical constraints is used to create a probability density function for the location of each atom in the protein.

1.1.2 First Approaches in Protein Structure Determination

Bioinformatics largely involves the characterization of protein structure and function, but it is not necessarily involved in the difficult task of detailed protein structure determination. There is a vast amount of protein sequence data available, and screening

these sequences would be greatly beneficial in terms of characterizing protein function (Panchenko et al., 2005).

Water plays a major role in biological function. The ability to correctly predict solvent accessibility from a protein sequence could be very useful in characterizing protein interaction and function as well as guiding protein modification and design. The results of the characterization of solvent-accessible surfaces are useful in many protein design and structural biology applications (Petrova and Wu, 2006; Pettit et al., 2007). This includes identification of catalytic and other key functional residues including those found on protein surfaces. This of course augments the restricted number of proteins with extensive 3D structures determined from X-ray and NMR. In addition, solvent-accessible surface prediction has garnered attention for its usefulness in the characterization of protein-protein interactions (Porollo et al., 2007). Furthermore, the characterization of solvent-accessible surface has been a standard first approach in determining protein structure. The ability to predict solvent accessibility of a given residue would yield great benefits.

Using the primary amino acid sequence to predict surface-accessible residues has been a standard first approach in structural biology's pursuit to model 3D protein structures. Solving this problem has been of great interest as a testing platform for a variety of machine-learning methods. Protein characterization from sequence data can also be applied to other biological issues such as identification of key core (e.g., strongly hydrophobic) residues (Berezovsky and Trifonov, 2001; Poupon and Mornon, 1999). Such methods hold the potential for increased understanding of the fundamentals of

protein folding. Methods utilizing sequence information usually use machine-training approaches and have shown accuracy of 70–78% (Adamczak et al., 2004; Richardson and Barlow, 1999; Rost, 1994;).

Currently, various methods have been reported to yield better prediction accuracy. Among these methods, the following are noteworthy: two-stage and related regression approaches, nearest neighbor method, decision tree methods such as random forest method, and support vector machine learning (Adamczak et al., 2004; Joo et al., 2012; Mizianty and Kurgan, 2012; Pugalenthi et al., 2012). Typically in these types of methods, the test (experimental) relative surface accessibility (RSA) value is predicted based on a particular or constraints model, and consensus predictors are generated via the use of a training data set.

1.2 Sequence Entropy

Another structural feature extensively used is the identification of the key core of proteins mostly made up of strongly hydrophobic residues (Berezovsky and Trifonov, 2001; Poupon and Mornon, 1999). The identification of key core residues can help define key constraints in modeling a protein's folding and characterization. Shannon entropies for protein sequences have been used to score amino acid conservation (Koehl and Levitt, 2002; Shenkin et al., 1991; Valdar, 2002). Sequence entropy (Shannon entropy) is the ability of a residue in protein sequence to mutate or change. The correlation of Shannon entropy is greater mutability at a particular amino acid position, the more able an amino acid is able to adapt to a mutation (Hseih et al., 2002). On one hand, the core (hydrophobic) regions are usually evolutionarily well conserved; hence,

they tend to have low sequence entropy values. On the other hand, the non-hydrophobic residues are usually solvent accessible, and tend to have higher sequence entropy values. The sequence entropy, E20, at some residue position k is expressed as

$$S_k = -\Sigma_{j=1,20} P_{Jk} log_2 P_{jk}, \quad (1.1)$$

here probability P_{jk} at amino acid sequence position k is derived from the frequency for an amino acid type j for N aligned residues. Here, each amino acid out of the 20 canonical represents individual groups.

There have been studies where sequence entropy information has been greatly helpful in characterizing protein-protein boundaries (Guharoy and Chakrabarti, 2005). Koehl and Levitt (2002) described a correlation between thermodynamic and sequence entropy in proteins. The Lustig group calculated two regions for sequence entropy and hydrophobicity of individual residues with respect to the inverse of their respective Cα packing densities (Liao, 2005). Out of these two regions, major region II corresponds to less than 11 Cα per 9Å radius and is principally flat and consistent with the most flexible residues. The majority of the most flexible residues display significant exposure to solvent.

The Lustig group previously reported the addition of a second term that uses the corresponding probability Pj of an amino acid type j to correct for random substitution that also does not significantly improve the noise or otherwise change the overall trends of the correlation plots (Liao, 2005). Although sequence entropy alone is not a unique identifier of structural features (Guharoy and Chakrabarti, 2005; Oliveira et al., 2002; Yan et al., 2006), it has shown some potential to illustrate protein-protein interfaces.

1.3 Overview

This thesis summarizes calculations that use a set of homology-based parameters to predict the RSA of protein residues. Thresholds were applied to the RSA values of a given residue where 1 was assigned as being on the surface and 0 as being buried. After reproducing and automating the existing training set of 268 protein sequences, a larger data set comprising 1363 diverse computationally designed proteins (described in Chapter 2) was incorporated in the calculations. BLASTP was used to align all of the proteins with known protein sequences from the database. These results were used to calculate sequence entropy and other homology-based parameters. Packing density was calculated for each residue in the 1363 data set. Entropy, functional parameters, and packing density were determined for each residue in the data set to create an aggregate training set of protein parameters. Once the final aggregate training set was created, statistical software, R, was used to perform linear regression for a total of 12 models comprising various combinations of the functional parameters and entropy values. This linear regression model employs a novel application of a query-based qualitative predictor in conjunction with quantitative protein sequence homology. The training set was used to generate coefficients for linear regression models; once these coefficients were generated, they were applied to the test set to generate estimated RSA values. Finally, as determined from X-ray 3D structure, accuracy was calculated for the prediction.

1.4 Organization of the Thesis

This thesis is organized into 6 parts: Introduction, Methods, Results, Discussions, Conclusion, and Future Studies. The second chapter, Methods, details the calculations and techniques used to come up with the results. This section also explains how linear regression was performed and how different functional parameters including entropy values were generated and used. Chapter 2 also contains a detailed description of how the data sets were generated. The third chapter, Results, includes the key results of the Methods sections. It contains various tables and figures that summarize outcomes of a number of experiments performed. The fourth chapter, Discussion, is focused on a review of available literature. This chapter also notes some areas where calculations could be improved. Chapter 5 provides concluding remarks regarding the methods and results and their context within the literature. Possibilities for future studies are described in the final chapter of this thesis, Chapter 6.

2. Methods

In order to study protein structure using sequence information, various homology parameters were calculated and their relationship with Cα packing density was investigated. In conjunction with protein sequence homology, a novel application of query-based qualitative predictors was then used to characterize solvent-accessible residues. Out of the three diverse sets of proteins, two (1363 and 268) were used as training sets, and the standard 215 (Naderi-Manesh et al., 2001) was used as a test set.

2.1 Protein Sets and Preparations

The 268 protein set, as the name suggests, is made up of 268 diverse protein chains (Mishra, 2010). The goal of recreating calculations and recharacterizing the 268 training set was first to automate the various steps involved. Previously, PDB IDs for the individual 268 training proteins were entered manually in the NCBI website. It was critical to be able to automate this process; with the development and incorporation of a larger training set (1363), it would be an extremely daunting and time-consuming task to manually run BLAST for of each the 1363 PDB IDs. There were instances where NCBI would take much longer (>5 minutes) to run BLAST manually for some PDB IDs. Additionally, it was essential to use the FASTA sequence corresponding to the PDB IDs as the header of the .txt output file for subsequent calculations of various sequence homology parameters and entropies. The automation of this process allowed for ease of incorporation of PDB IDs for additional training and test sets and reduced the number of manually introduced errors.

The BLAST output file automation was made possible by the python script *download_blast.py*, and the FASTA sequence was downloaded with the aid of *download_blast.py* (see Appendix A). Each of these programs can be run in batch, and the resulting BLAST and FASTA files are saved in an output folder.

The 215 set was used as the test set and was the standard 215-protein list from Naderi-Manesh et al. (2001). Earlier calculations of surface-accessible residues have been noted in our previous paper (Rose et al., 2011). The largest set of the three protein sets, 1363, was used as training data. Bondugula et al. (2011) listed a diverse set of 6511 protein domain with 50 designed sequences per domain. The proteins are designed using computational sequence design methods to engineer proteins with desired properties such as increased thermal stability and novel functions.

The 6511 proteins were available in SCOP (Structural Classification of Proteins) format (Bondugula et al. 2011). Since all of the programs and scripts written worked with PDB IDs, the SCOP identifiers of the 6511 set were converted to PDB IDs. SCOP identifiers are made up of 7 characters; the first character is "d," the subsequent 4 letters stand for the PDB name, the third part is the DB chain id ("_" if none, "." if multiple), and finally a single-digit integer, if needed, uniquely specifies the domain ("_" if not). All the SCOP identifiers were converted using a python script called *SCOPid_to_PDBid.py*, which is listed in Appendix A.

Each protein had to be entered in the PDB website to validate its identifier. Again, a python script called *common pdbs in the Bondugula set vs PDB library.py* (see Apendix A) was written to address this issue. Steps 1 to 4 in Figure 2.1 describe this

validation method. Out of the original list of 6511 proteins, only 5157 were found in the pbd library, and thus the remaining 1354 PDB names were dropped from the list.

The list of 5157 proteins (see Figure 2.1) was entered in a protein culling service called PISCES (Wang and Dunbrack, 2003). The 5157-protein set was culled for PDB chain identifiers that share $\leq$ 25% identity, have a structural resolution of 0.0 to 2.5 Å, an R-factor of $\leq$ 0.3, and a sequence length of 40 to 10,000. Protein chains with Cα-only entries were eliminated, and only protein structures determined via X-ray crystallography methods were selected. The same culling standards were used on the 268 and 215 data sets as well (Mishra, 2010). The final list of all the PDB IDs that that were part of the 1363 training data set is presented in Appendix B.

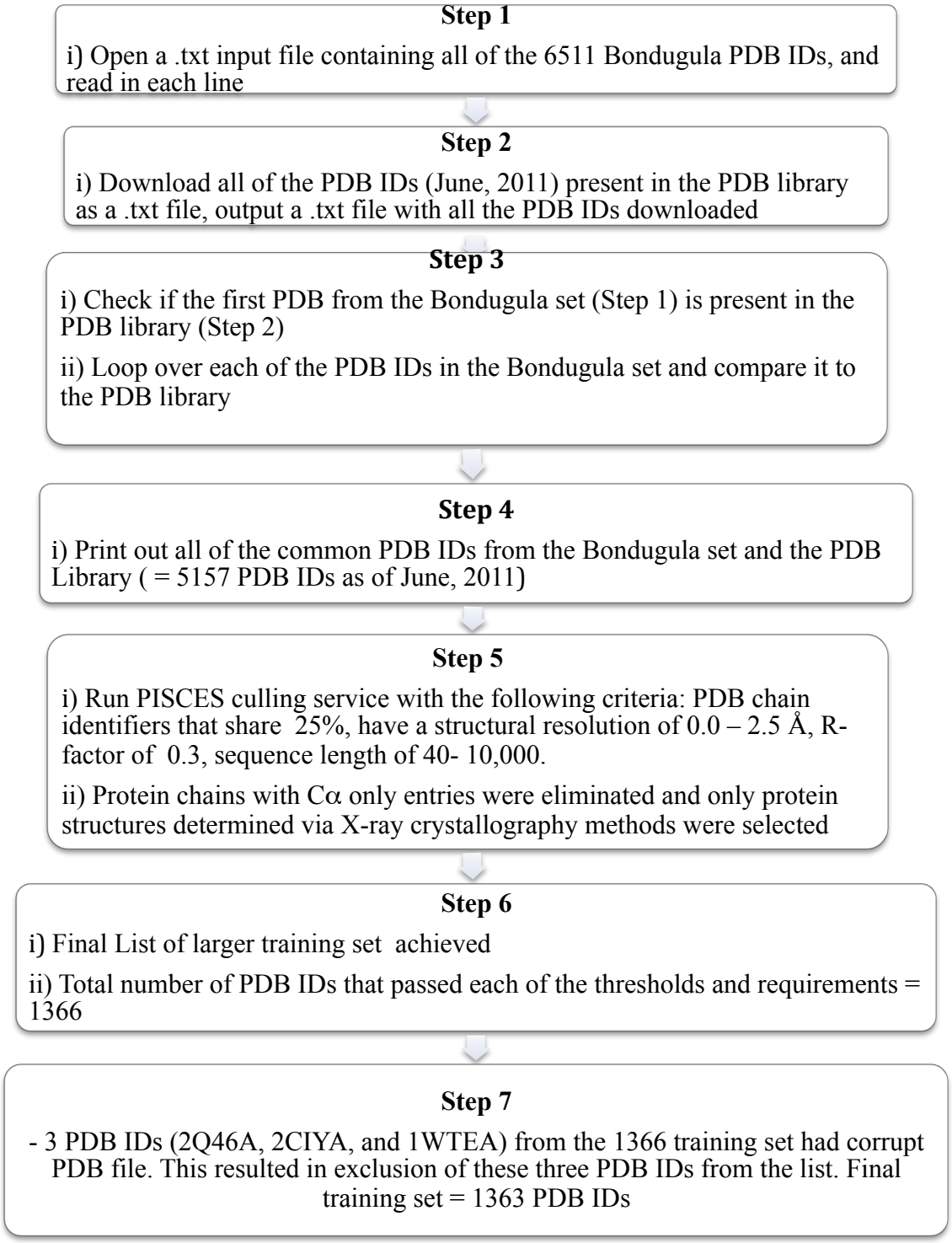

Figure 2.1 Flowchart of steps involved in the generation of the 1363 training data set based on the list of proteins from Bondugula et al. (2011).

2.2 Residue Packing Density

Residue packing density calculates packing density in a residue's native state and is used to measure protein compactness. It is calculated by using X-ray determined Cα coordinates of a given query protein. To calculate protein residue packing density, mmCIF of each protein in the protein sets were downloaded from RCSB PDB (2011).

X-ray crystallography is used to determine the atom co-ordinates information of a protein. Once all the mmCIF files were downloaded, all the Cα coordinates were extracted at each residue position using the python code *calculate_density.py* (see Appendix A). This program calculates density between any two residues using the following equation:

$$\text{Dist (i, j)} = \sqrt{x(i) - x(j)^2 + (y(i) - y(j)^2 + (z(i) + z(j)^2} \qquad (2.1)$$

where x, y, and z are the Cα coordinates at that sequence position. Next, the number of Cα atoms within the radius of 9Å around the residue of interest is calculated. Finally, the packing density at that residue was calculated by determining all other Cα residue positions within 9.0 Å from the Cα position of record.

In earlier work conducted by the Lustig group (Mishra, 2010), all the density values were calculated using PERL scripts. In this thesis, all density values were calculated via python scripts using similar logic and calculations. The python program *download_mmCIF.py* (see Appendix A) downloads each of the mmCIF files for the corresponding PDB ID listed in the input file. On one hand, the mmCIF has some residues whose coordinate values are unavailable; these values were assigned NA. On the other hand, packing density equal to 0 was assigned to the unknown residue such as

'X'. Both NA and 0 density values were excluded from frequency plots and correlation plots.

2.3 Sequence Variability

Sequence variability for each residue was measured by sequence entropy or Shannon entropy. Sequence entropy is defined as the measure of disorder or randomness in a system. A list of PDB names was acquired for all the protein data sets. Each PDB name was entered in the Basic Local Alignment Search Tool (BLAST 2.2.18+), a protein database program provided by the National Center for Biotechnology Information (NCBI). BLAST searched all the databases available for non-redundant protein sequences using a BLOSUM62 matrix and default gap penalties for each mutational insertion or deletion. Once the PDB name of a protein of interest was entered in the BLASTP website, it is referred to as the query sequence. The query sequence is compared to all of the sequences in the database, referred to as the subject, that are evolutionarily similar to it. The search was performed using default settings except for the Max target sequences setting, which was altered from 100 to 10,000. The aligned residues were extracted from BLASTP results using a python script labeled *blast_to_entropy.py* (see Appendix A). BLASTP alignments with bit scores equal to 40% of the highest bit score were only used for entropy calculations. As noted in a previous thesis, a 40% cut off seemed to provide an ideal balance between homology and the diversity of sequence variability (Yeh, 2005).

Alternative calculations for sequence entropy were also applied, one involving an application of a 6-term sequence entropy (Mirnya and Shakhnovich, 1999) to the existing

alignments for all of the data sets. The E6 entropy groups amino acids into six groups and is calculated by:

$$s_l = -\sum_{i=1}^{6} p_i\,(l) \log p_i \tag{2.2}$$

where p_i is the frequency of each of the 6 classes i of residues at position l in multiple sequence alignment. The 6 classes of residues are aliphatic (AVLIMC), aromatic (FWYH), polar (STNQ), positive (KR), negative (DE) and special (GP). Once the PDB name of the protein of interest was entered in the BLASTP website, it was referred to as the query sequence. The query sequence was compared to all the subject sequences available in the database that are evolutionary similar to it. The search was performed using default settings except for the Max target sequence setting, which was altered from 100 to 10,000. The aligned residues were extracted from BLASTP results using a python script labeled *blast_to_entropy.py* (see Appendix A).

BLASTP alignments with bit scores equal to 40% of the highest bit score were used only for entropy calculations. A 40% cut off seemed to provide an ideal balance between homology and the diversity of sequence variability (Mishra, 2010).

2.4 Homology-Based Parameters

The development of homology-based parameters was one of the first approaches used to predict solvent accessibility of residues. Once the training and test sets were selected, BLASTP (2010) from Genbank was used to align the sequences. The protein of interest, the query sequence, is aligned with other homologous subject sequences in the

database, subject sequences that are closely related. Sequence homology parameters for the Lustig group used are 20-term entropy (E20), 6-term sequence entropy (E6), aligned residues that are strongly hydrophobic (FSHP), and aligned small residues (FSR).

Fraction strongly hydrophobic (FSHP) uses strongly hydrophobic residues, VILFYMW (Poupon and Mornon, 1999). FSHP is calculated in the following manner:

$$Fraction\ strongly\ hydrophobic_i\ (FSHP) = \frac{Number_{SHP_i}}{Total\ Number\ of\ aligned\ Residues_i} \quad (2.3)$$

Where, $Number_{SHP_i}$ is the number of strongly hydrophobic residues at sequence position i. Fraction small residues (FSR) refer to residues Gly or Ala and are calculated as follows:

$$Fraction\ small\ residues_i\ (FSR) = \frac{Number_{SR_i}}{Total\ Number\ of\ aligned\ Residues_i} \quad (2.4)$$

Fraction Alanine residues (FA) refer, as their name suggests, to residue Ala and are represented by:

$$Fraction\ Alanine\ (FA) = \frac{Total\ number\ of\ Alanine\ residues}{Total\ number\ of\ aligned\ residues} \quad (2.5)$$

Fraction Glycine residues (FG) represent GLY residues and are represented by:

$$Fraction\ Glycine\ (FG) = \frac{Total\ number\ of\ Glycine\ residues}{Total\ number\ of\ aligned\ residues} \quad (2.6)$$

2.5 RSA Calculations

In this work, residue RSA is calculated with the bioinformatics tool called NACCESS (Hubbard and Thornton, 1993). The NACCESS program calculates accessible surface by rolling a probe of a given size around a van der Waals surface. It also determines a residue accessibility file (.rsa) containing summed atomic-accessible surface areas over each protein residue. The program also normalizes the accessibility of

each residue calculated as the percent of accessibility compared to the accessibility of that residue type in an extended A-x-A tripeptide format (Hubbard et al., 1991).

The NACCESS RSA values were used as a standard to compare predicted RSA values generated from calculations for this research. RSA values can range from 0 to, very occasionally, 150. Anything higher than or equal to 20 is regarded as being on the surface and anything less than 20 is considered buried (Carugo, 2000). A binary system was incorporated to support the calculations pertaining to this research. Any RSA value greater than or equal to 20 was assigned a 1, and RSA values less than 20 were assigned a 0 and labeled as buried. Programmatically, the following python scripts were used to calculate RSA values: 1. *download_pdb.py*, 2. *run_naccess.py*, 3. *RUNNACCESSonUnix.pl*, 4. *extract_data.py*. The corresponding python scripts can be found in the Appendix A section of this thesis.

2.6 Determination of Qualitative Predictors

Linear regression is a method used to model relationships between a scalar variable Y and one or more variables denoted as X. In this method, data are modeled using linear functions, and unknown parameters are estimated from the data. There are two main kinds of variables used in regression analysis: quantitative variables and qualitative variables. On one hand, quantitative variables are expressed as numerical values. Qualitative variables, on the other hand, are categorical expressions. For example, the corresponding barometric height of mercury for a reaction chamber would be 153 mm, whereas pressure could be classified qualitatively as either high or low. Qualitative predictors have been widely used in the social sciences and related fields

(Hellevik, 2009), but have been relatively unexplored in molecular science. This novel method was used to predict RSA values in this research.

In this section, a simple calculation expressing the application of query-related qualitative predictors for RSA prediction will be illustrated. One of the ways to quantitatively express categorical information is to use indicator variables that take on values, or 0 and 1 (Kutner et al., 2004). Two such delimeters will be expressed here: strongly hydrophobic residues (SHP; VILFYMW) and the remaining non–strongly hydrophobic residues (NSHP). The general model for a first-order linear regression model is:

$$Y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \varepsilon_i \quad (2.7)$$

where Y_i is a straight line with intercept β_0 , slope β_1, and ε_i as residual error function (Kutner et al., 2004). For the SHP and NSHP regression calculations, the 73,675 residue RSA values are fit to the variable X_{i1} corresponding to the E6 value at each residue, i. Here, the two qualitative predictors are SHP (X_{i2} is 0) and NSHP (X_{i2} is 1). The generalized response function can be expressed as:

$$E\{Y\} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \quad (2.8)$$

Substituting the SHP and NSHP value, the fit equation for hydrophobic and non-hydrophobic equation becomes:

$$E\{Y\} = \beta_0 + \beta_1 X_1 \text{ where } X_2 \text{ is } 0 \quad (2.9)$$

$$E\{Y\} = (\beta_0 + \beta_2) + \beta_1 X_1 \text{ where } X_2 \text{ is } 1 \quad (2.10)$$

The regression for these equations was generated using the programming language, R. Figures 2.2 and 2.3 show the fit for this calculation. To create linear regression plots

with qualitative predictors, the training data sets' .csv files were used as input files. The input files include one column for NACCESS RSA values and another for the corresponding E6, E, FSR and FSHP values. Now our model can include 20 qualitative predictors that are associated with each amino acid type (for example, A) for all the sequence residues.

2.7 PSI-BLAST Calculations

The NCBI website was used to perform PSI-BLAST searches. The PDB names were entered in the search field, and this time around PSI-BLAST was chosen under the algorithm section instead of blastp. Under Algorithm parameter, "Max target sequences" was altered from 500 to 10,000. After each search result was presented, a second and third iteration were performed.

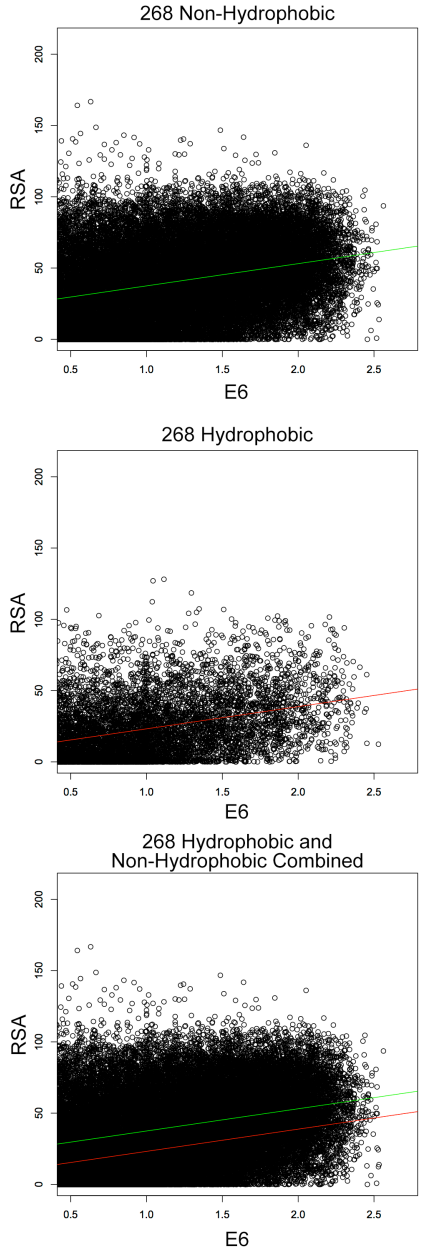

Figure 2.2. Sample regression fit for 73,734 query residues from the 268 training data set. Here, the NACCESS RSA values to a variable term X_{i1} as E6 and the qualitative predictor terms having two values, where X_{i2} is 0 (top) for strongly hydrophobic (SHP) query residues and X_{i2} is 1 (middle) for non–strongly hydrophobic (NSHP) residues, are presented. The slope, 15.6, corresponding to the variable term is the same for both plots, while intercepts are 7.6 and 21.9 for β0 and (β0 + β2), respectively. The aggregate plot is shown (bottom).

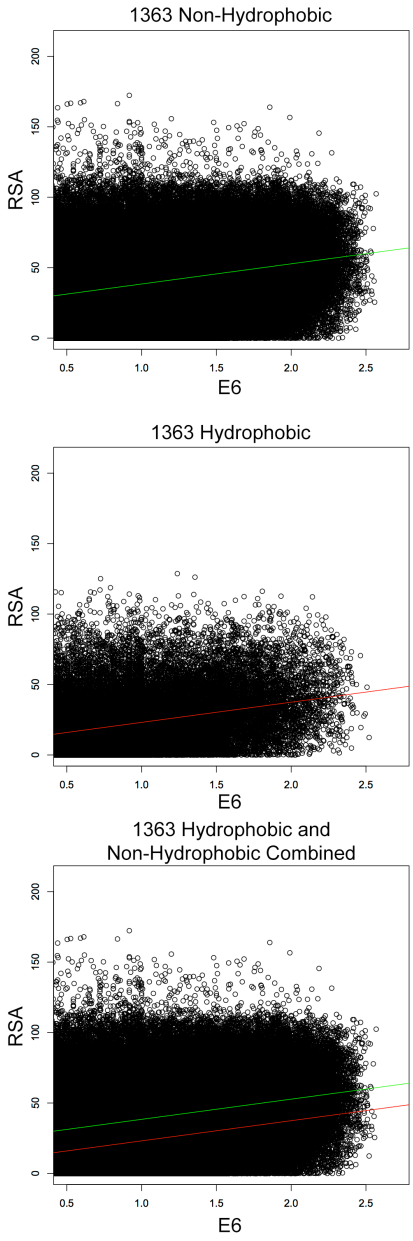

Figure 2.3: Sample regression fit for 319,551 query residues from the 1363 training data set. Here, the NACCESS RSA values to a variable term X_{i1} as E6 and the qualitative predictor terms having two values, where X_{i2} is 0 (top) for strongly hydrophobic (SHP) query residues and X_{i2} is 1 (middle) for non–strongly hydrophobic (NSHP) residues, are presented. The slope, 14.3, corresponding to the variable term is the same for both plots, while intercepts are 8.8 and 24.1 for β0 and (β0 + β2), respectively. The aggregate plot is shown (bottom).

2.8 Accuracy Calculations

The accuracy for both buried and surface-accessible residues was then calculated by the standard expression of Richardson and Barlow (1999):

$$\text{Accuracy} = \frac{Number\ of\ assignment\ to\ correct\ category}{total\ number\ of\ assignment\ made} * 100\ \% \qquad (2.11)$$

To calculate accuracies for each of the models, principally linear regression was applied to each of the 13 main models used to generate predicted RSA values. These models were made up of various combinations of the two types of entropy (E and E6), and homology-based parameters (FSR, FSHP) with amino acids were used as qualitative predictors. The y-intercept for the line of best fit determined the threshold for the predicted RSA. With the y-intercept for hydrophobic versus non-hydrophobic linear regression line of best fit, a threshold of >23 was classified as surface, whereas ≤23 was classified as being buried. This is a result of internal optimization of the test results. The accuracy equation above was then applied in a binary fashion. Any NACCESS RSA that was ≥20 was assigned as 1 meaning on the surface (while 0 was for residues that were buried), any predicted RSA that was >23 was also assigned as a 1. Next, it was noted whether a given residue matched as being on the surface or buried when compared to the NACCESS RSA and predicted RSA. Finally, accuracy was discovered by dividing the number of correctly assigned residues by the total number of predictions made and reported as a percent. Programmatically, the generation of linear regression models, the predicted RSA and accuracy calculations were made possible by R program code *accuracy.R* (see Appendix A)

2.9 Aggregate Analysis and Correlation Plots

Once correctly aligned, files were obtained for each of the proteins in the protein sets and additional calculations were conducted. Comparable to procedures listed by Mishra (2010), at each density position the different homology parameters and entropies were averaged. This averaging at each density value was obtained with the help of the python script *extract_density_frequency.py* (see Appendix A). For example, at the density 4 average, all of the E values for residues that have a density of 4 are present. Similarly, averages were filled out for each of the entropies, FSR, and FSHP for each of the density values. The python script was used to generate a table in .csv format with all the average values. Finally the .csv files were converted to .xlsx format. These averages were used to generate different correlation plots by plotting various homology-based parameters against inverse density.

2.10 Frequency Distributions

The homology-based parameters (E20, E6, FSHP, FSR) were aligned together properly together by matching each of the density values with corresponding residue positions. These homology parameters for each of the proteins in a given training or test set were then compiled into a single .csv so that frequency distribution histograms could be generated. As noted in previous work by the Lustig group (Mishra, 2010), each of the density values equal to 0 and NA were eliminated from the list.

Query length for each of the proteins in the lists was calculated with the python script *extract_query_length.py* (see Appendix A). This program used BLAST output .txt files as input to extract query length. The length of alignment was also generated with

the help of another python program, *extract_record_length.py*. For the bit score frequency plot, the frequency of subject proteins at BLAST bit score were generated with the help of *extract_bit_score.py*.

2.11 Assimilation of Additional Methods to Improve Accuracy

2.11.1 Incorporation of Tertiary Protein Structure Information

A second-stage prediction method was the use of tertiary protein structure information to study its impact on prediction accuracy. The goal of this research was to study whether protein tertiary structure information from a limited subset of proteins can aid in assigning the solvent-exposed residues of a protein outside the subset. Previous work from the Lustig group at San Jose State University investigated tertiary contact information (Nguyen, 2012). As outlined in his thesis, a protein tertiary contact is defined as a pair of amino acid interactions that are separated by at least 10 residues in the protein primary sequence (Kallblad and Dean, 2004). The atomic distances of these two amino acids need to be less than the sum of the van der Waals radii of the 2 atoms plus 1.0 Å (Kim and Park, 2003). Protein tertiary structures are also critical for protein stability; while secondary structures are usually unstable, tertiary interactions make them more stable (Daggett and Fersht, 2003).

It has been shown that tertiary interactions in a protein are usually buried, well conserved, and more densely packed than other protein residues (Do, S. and Lustig, B. San Jose State University, San Jose, CA. Unpublished work, 2010). Furthermore, tertiary contact information suits secondary protein structure prediction very well in terms of

sequence entropy, packing density, and RSA values. Therefore it makes sense to utilize tertiary contact information as a second filter in RSA prediction.

Previous research noted that, out of the 268 proteins in the training set, 75 were known to have tertiary contact (Nguyen, 2012). A 95% confidence interval was applied to the 75 tertiary contact proteins to derive the appropriate tertiary contact threshold. The 95% confidence interval here implies that, out of all the tertiary contacts presented, 95% (including false positives) of the lowest-threshold tertiary contact sequence entropy values are correctly predicted as being buried within a protein. This is because tertiary contacts are more conserved when compared to other residues, and most likely to be found buried. Once the threshold was established, it was applied to the entire 215 test set. This calculation was carried out by dividing each of the proteins into a separate .csv file that included all of its residues. For example, the first protein in the 215 set is 119LA, which has 162 residues; therefore 119LA.csv would have 162 residues present in it. Following this, a matrix was created. In this example, the matrix was 162 by 162. For each possible position in the matrix, entropy was averaged for the two residues involved (column residue and row residue). If the average entropy of any two residues was greater than the threshold value, then those two residues were predicted to be on the surface, denoted by a 1; otherwise, the position was predicted to be buried and assigned a 0. This information was applied to the predicted RSA values obtained from linear regression data. Anytime, a given residue position was predicted to be on the surface in the tertiary contact matrix, the linear regression data was altered to match the prediction of the tertiary filter. Performing the calculations manually—checking each and every residue

assigned a 1 in the matrix and referring back to the prediction Excel file— would have been not only challenging, but also prone to many errors. Thus, to address this issue, two R scripts were written: *tertiary_contact_filter2.R* and *accuracy_with_filter_2.R* (see Appendix A). The first program, *tertiary_contact_filter2.R,* takes entropy data from multiple residues of a given protein and creates a filter matrix. The second program, *accuracy_with_filter_2.R,* imposes the threshold to each matrix value and recalculates solvent accessibility accuracy. Contrary to our initial hypothesis, the incorporation of tertiary information in the prediction model did not improve accuracy.

2.11.2 Additional Models Applied

Upon further investigation of individual protein accuracies, it was noticed that the smallest residues, alanine and glycine, were the most mispredicted amino acid residues. The small residue fraction was represented as a model in conjunction with the 20 amino acids as a qualitative predictor. However, the initial 11 models were missing fraction strongly hydrophobic (FSHP) in combination with amino acids as qualitative predictors. FSHP + AA was added as a new model. The results of this calculation are presented in Table 3.4 in the Results Section of this thesis.

Fraction Alanine (FA) was the first tested on the regression models. For the new model, FSR was replaced with FA. The results of this test are presented later in this thesis. Again, comparable to FA, model FG also replaced FSR as a model, and accuracy for all the models with FSR replaced with FG were recalculated. The results of this calculation are presented in Table 3.4. Finally, it was not sufficient to just substitute FSR

with either FG or FA. As a third test to this principle, FSR was swapped with FG + FA; these results are also presented in Table 3.4.

2.11.3 Incorporation of a Categorized Protein Data Set

An additional approach that incorporated a new data set was used to study any further impact on the accuracy calculations. In a previous study conducted by Pettit and co-workers (2007), a set of 618 proteins categorized to 15 different chemical groups was investigated by a HotPatch study. HotPatch is a statistical analysis system that finds unusual patches on the surface of proteins and computes just how unusual they are (called patch rareness), as well as the functional importance of each patch. The set of 618 proteins are divided into 12 different groups: proteases, hydrolases, kinases, transferase, oxidoreductases, catalytic general, DNA/RNA interacting, negative ion binding, small-molecule interacting, carbohydrate interacting, lipid interacting, and positive-metal ion binding. An individual PDB ID from the 618 set could belong to any one of these groups or multiple groups. For the purpose of this study, the 12 original groups were split into three simpler groups: oligomers, protein binding, and generic without oligomers and protein binding. Notably protein binding refers to proteins that transiently bind to other proteins.

The main goal of this experiment was to implement the group information to both the test set and the training set to observe if any progress would be made in the accuracy numbers. There were multiple sub-sets of calculations carried out with this principle, all of which are described in the following subsections of this thesis.

The first group of proteins investigated was protein binding. The PDB IDs from the 268 training set, 1363 training set, and 215 test set all shared common PDB IDs with the 618 set. A group of PDB IDs from the 268 set that overlapped with the protein binding group was selected; similarly, the PDBs in common between the 215 and 618 protein binding groups were also chosen.

3. Results

3.1 Characterization of Protein Lists

The characterization of protein lists based on the method of structure determination, resolution R-factor, free R value, protein length, and alignment length for the 268 training set has been presented in an earlier study (Mishra, 2010). For the 268 training set, frequency distributions and correlations found in the previous studies were extracted from manually derived BLAST output files. In this section, results from the automated BLAST output files are presented for comparison and validation purposes. The newly developed training set, 1363, has also been characterized using frequency and correlation plots in this section.

The automated BLAST-generated frequency distributions for the characterization of the 268 training set are shown in Figure 3.1. A frequency plot of length of query protein length with regard to the frequency of occurrence is presented in Figure 3.1A. The highest and second highest peaks are represented by a frequency of 39 at a 350-protein length and a frequency of 38 for 200-protein length. The histogram distribution appears to be distorted and skewed right; the right tail of the graph is considerably longer than its left tail. The mean value of this histogram was at 283, and the mode was presented at 340. The length of protein ranges from 50 to 950; however, the majority of the proteins (95.5%) have lengths between 100 and 600.

Figure 3.1B displays a histogram for density of query residues as a function of frequency of query residues. The mean and mode for this distribution are 17.5 and 15, respectively. The histogram has normal distribution, with 95.6% of data points appearing

within the intervals of 7 and 22. The highest frequency of query protein density, 6467, occurred at a density value equal to 15. This corresponds to a normal Gaussian-like distribution.

The number of alignments as a function of frequency of query residue distribution plot is presented in Figure 3.1C. The number of alignment ranges from 200 to slightly over 2000 for the frequency of query proteins. The maximum for the number of alignments is at 1200 alignment for the frequency of 104. The shape of the distribution is skewed slightly left, with 69.0% of alignments hovering from 1000 to 1200. The mean, mode, and median of the distribution are at 962.55, 1000, and 1000, respectively.

The distribution plot of the BLAST bit score (Figure 3.1D) displays a right-skewed pattern, as expected for such a distribution (Liao et al., 2005). The maximum BLAST bit score occurs at the value of 1894, with the minimum at 29. The bit score of 100 is the highest, occurring at the frequency of 88,198. The mean BLAST bit score is 224, the mode is 37, and the median is at 164.

The goal of re-characterizing the 268 training set was to compare and validate the automated BLAST-generated output performed in previous work by the Lustig group (Mishra, 2010). All four of the distribution plots (length of proteins, density, number of alignments, and BLAST bit score) are almost identical to the manually generated BLAST output calculations. Both sets of the distribution plots have the same maxima, minima, general trend of the histogram, and distribution. This validates the reliability of the python program *download_blast.py* to automatically download BLAST files from NCBI

website and shows that the outputs are comparable to manually downloaded BLAST files.

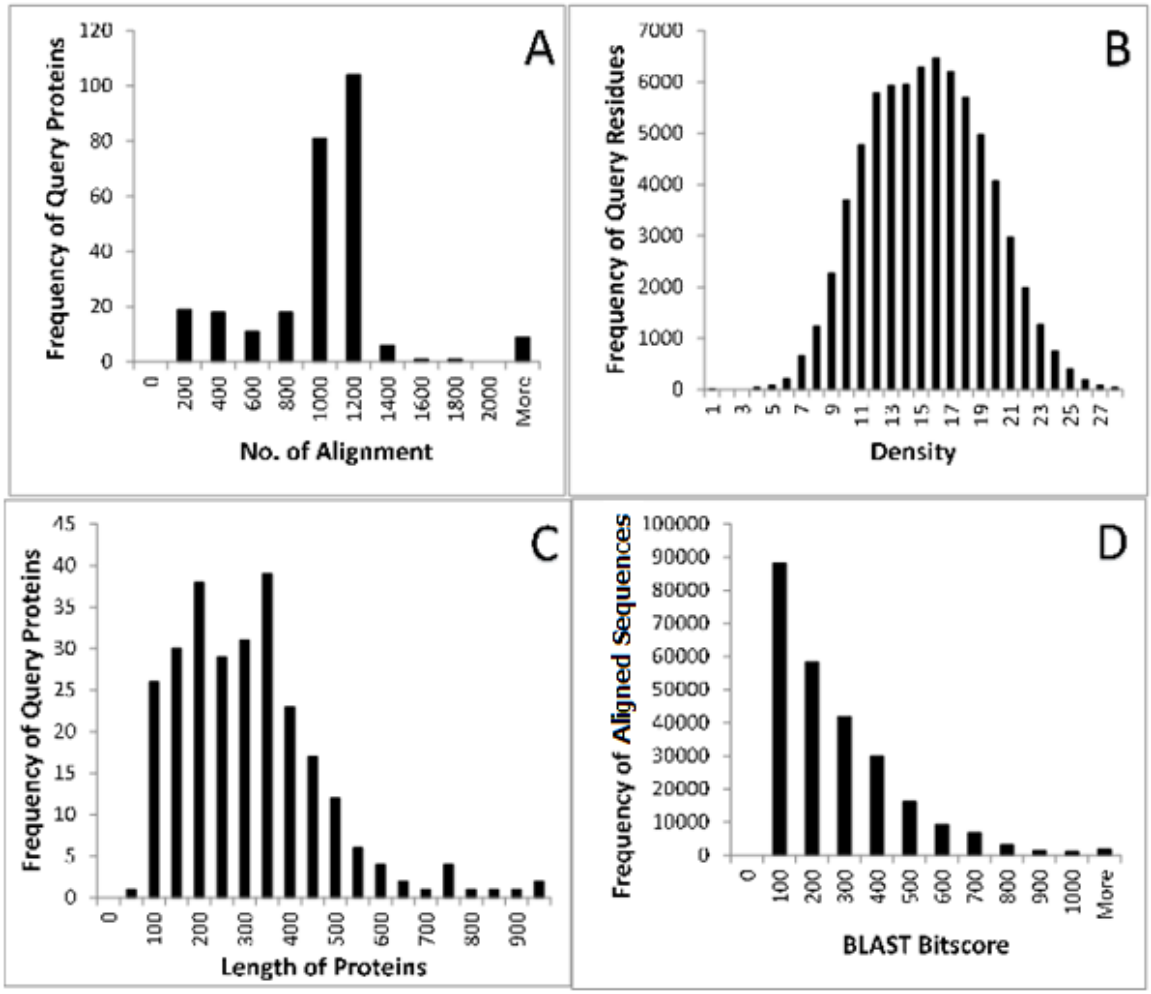

Figure 3.1. Frequency distributions for the characterization of the 268 learning set list of proteins. The 268 proteins in the list have a total of 73,734 query residues, and a total of 257,963 aligned subject protein sequences were used for these calculations. A. Frequency of query residues with respect to length of each protein of the 268 learning set. B. Frequency of 73,734 query residues with respect to each packing density. C. Frequency of query proteins was plotted against a number of alignments obtained from NCBI BLASTP outputs for the learning set. D. Frequency of 257,963 aligned subject sequences with respect to BLAST bit scores.

Figure 3.2 displays the frequency characterization of the 1363 training set. Since the BLAST output for the manual download and automated download were similar, 1363 BLAST files were automatically downloaded with the aid of *down_blast.py*. The frequency plot for the 1363 training set (Figure 3.2A) also has length of query protein with respect to frequency of query proteins. This histogram displays a slightly left-truncated normal distribution. The most frequent query protein length at 150 is shown at a frequency of 281. The mean of the query length is at 243.8, and the mode of the distribution is 129. Comparable to the protein lengths of the 268 training set, the protein length for the 1363 training data set also ranges from 50 to 950, and the majority of proteins (95.5%) have lengths between 100 and 550.

The frequency of query residues versus density histogram (Fig 3.2B) displays a Gaussian-like distribution. Density of 14 seems to be the most frequent density value at 50,833; however, densities 12 and 16 also flag as close values at 50,024 and 50,457, respectively. It is also evident from the figure that the majority, at 96.2%, of data points occur at densities between 8 and 23.

The number of alignments associated with the query proteins of the training set list (Figure 3.2C) ranges from 0 to 2000 or more. The maximum number of alignments occurs with a frequency of 391 at a sequence alignment of 1000 for 1KJQA. The bit score distribution plot is right-skewed, as expected (Liao et al., 2005). It indicates a BLAST bit score maximum at 100 occurring 500,000 times.

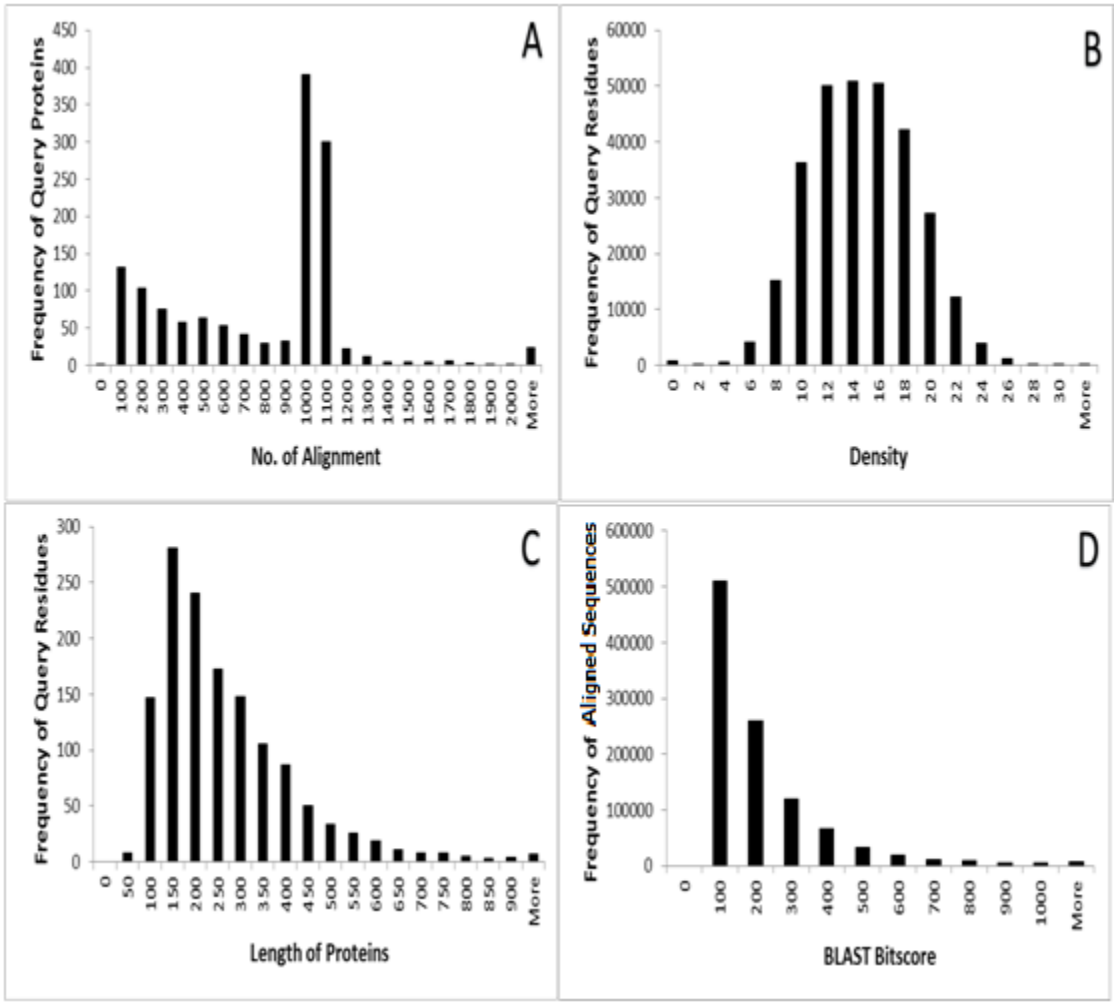

Figure 3.2. Frequency distributions for the characterization of the 1363 learning set list of proteins. The 1363 proteins in the list has a total of 319,551 query residues and total of 1,055,920 aligned subject protein sequences were used for these calculations. A. Frequency of query residues with respect to length of each protein of the 268 learning set. B. Frequency of 318,840 query residues with respect to each packing density. C. Frequency of query proteins was plotted against a number of alignments obtained from NCBI BLASTP outputs for the learning set. D. Frequency of 1,055,920 aligned subject sequences with respect to BLAST bit scores.

Figure 3.3 represents the histograms for frequency distributions for entropies and homology-based parameters of the 268 training set of proteins. A total of 73,724 query residues and a total of 257,963 aligned subject protein sequences were used. Figure 3.3A presents the distribution of the calculated 20-point entropy (E20) value as a function of frequency of query proteins. An entropy value of 0 occurred at the highest frequency, with a total of 10,239 residues having this entropy value. The average entropy value for the residues was 1.10, while the median was 0.972. The histogram indicates multimodal characteristics. The maximum calculated entropy value was 3.895. The lower entropy value represents a well-conserved residue. The fact that more than half (51.3%) of the residues have entropy values between 0 and 1 indicates that half of the residues are well conserved in the 268 training data set.

The distribution of 6-point entropy (E6) as a function of frequency of query residues is presented in Figure 3.3B. Similar to E20 distribution, the highest frequency of entropy values were observed at E6 = 0. However, the average E6 value at 0.64 was much lower than the E20 average (1.1). The median for this distribution was at 0.397 E6 value. Comparable to the E20 distribution plot in 3.3A, the E6 distribution also appeared to be multimodal. The maximum E6 value was 2.562.

Figure 3.3C presents the frequency of fraction small residues (FSR) as a function of frequency of query proteins. The mode of this distribution was at FSR = 0. The average FSR calculated value was 0.1637, while the median FSR value was 0.006. The FSR value of 0 occurred at the highest frequency of 31,420 residues. The FSR distribution plot is also right skewed, with slight increases in frequencies of 1, 1.1, and

>1.1. Finally, Figure 3.3D is the frequency plot of fraction strongly hydrophobic residues (FSHP) as a function of frequency of query residues. The mean, mode, and median of this distribution were 0.3247, 0, and 0.0271774, respectively. Here 59.06% of residues observed have FSHP values between 0 and 0.1. For the distributions in Figures 3.3C and 3.3D, there are clear decoupled components, which has been consistently noted with the manually generated BLAST output calculations (Mishra, 2010) and automated 268 distributions.

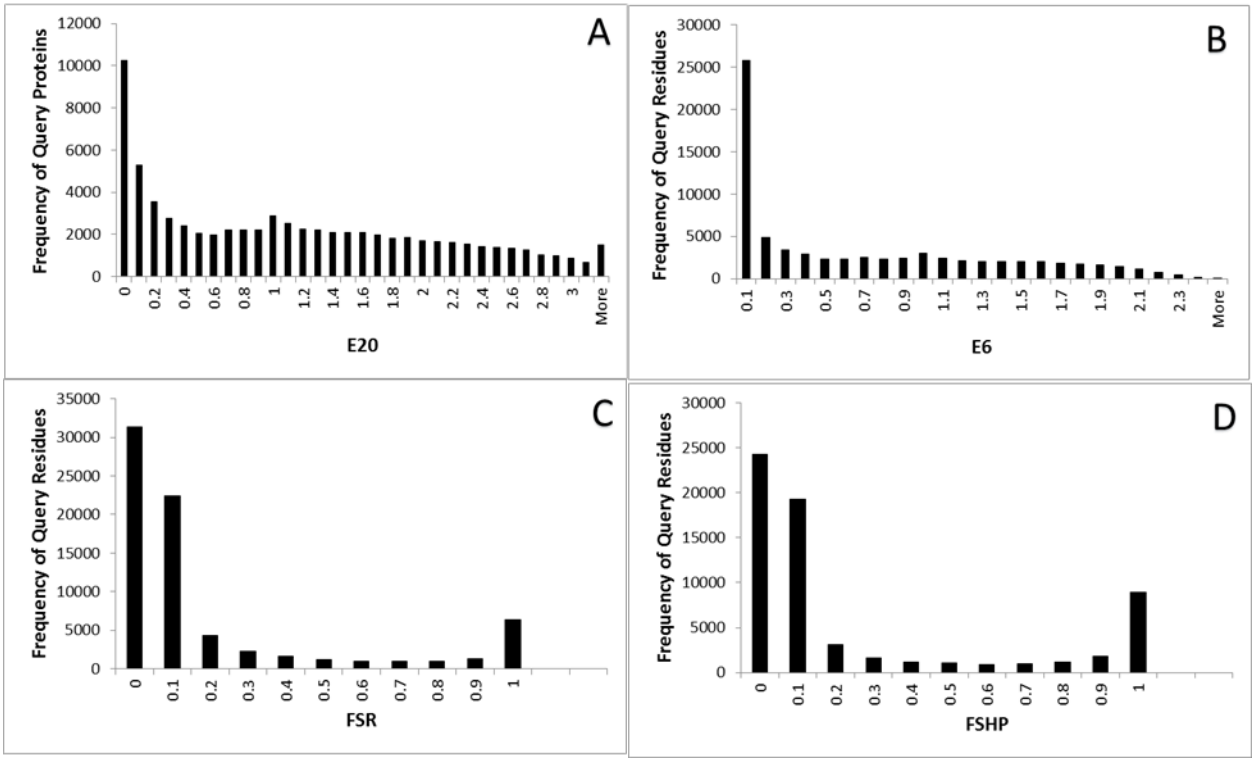

Figure 3.3. Frequency distributions of entropies and homology-based parameters of the 268 training set list of proteins. The 268 proteins in the list has a total of 73,734 query residues and total of 257,963 aligned subject protein sequences were used for these calculations. A. Frequency of query residues with respect to entropy values of the 268 learning set. B. Frequency of query residues with respect to E6 values of the 268 learning set. C. Frequency of query residues with respect to fraction small residues (FSR). D. Frequency of query residues with respect to fraction strongly hydrophobic residues (FSHP)

Figure 3.4 shows the frequency distributions of entropies and homology-based parameters of the automated 268 training set for the two major regions, Region I and Region II. Figure 3.4A presents the E20 distribution for the two major regions, and 3.4B presents the E6 distributions for the two regions. It is observed that at lower entropy (0 and 0.3) the majority of residues have RSA value less than 20, whereas at higher entropy

most of the residues have RSA ≥ 20 for both E20 and E6 distributions. The frequency distribution of FSR for the two regions peaks at low FSR value (0 and 0.1) and high FSR values (1 and 1.1), while the distribution flattens out in the middle ranges. This indicates that FSRs are found both in the core and surface of the protein. The highest frequency are observed at FSR of 0.

Finally, Figure 3.4D presents the distribution of the 268 training set for the two major regions and shows bimodal characteristics. An FSHP value of 0 indicates the absence of strongly hydrophobic residues. The majority of residues with FSHP = 0 correspond to an RSA ≥ 20; this was expected because during protein folding most of the hydrophobic residues are buried. When the FSHP = 1 or 1.1, the majority of the query residues have an RSA value of less than 20. The findings and trends of the automatically generated BLAST output homology parameters showcase similar trends as reported for the manual BLAST output calculation (Mishra, 2010).

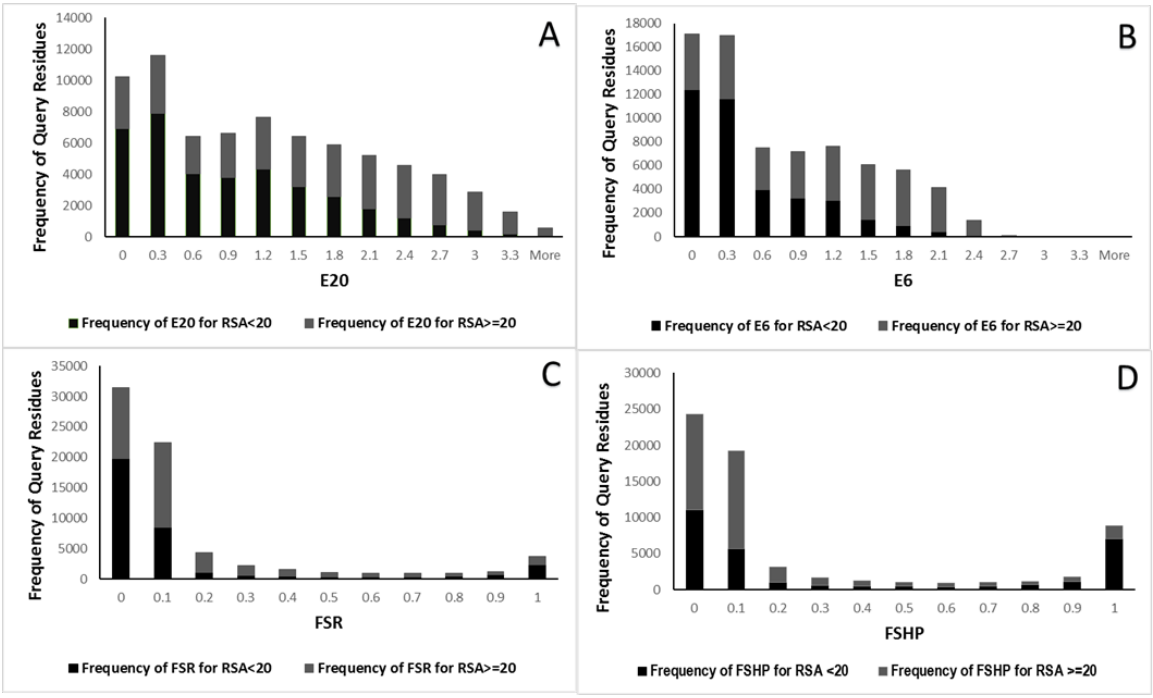

Figure 3.4. Frequency distributions of entropies and homology-based parameters of the 268 data set for the two major regions, Region I and Region II. A total of 73,734 residues were divided into Major Region I (RSA <20) and Major Region II (RSA ≥ 20). A. E20; B. E6; C. FSR; D. FSHP.

The histograms for E20, E6, FSR, and FSHP residue occurrence pertaining to the 1363 training set are presented in Figure 3.5. A total of 319,551 query residues were used to generate the distribution histograms. In Figure 3.4A, E20 is plotted as a function of frequency of query residues; the maximum E20 value is at 3.942, and the minimum is at 0. The distribution of E20 appears to form two clusters, and this suggests two separate normally-distributed populations. The highest frequency for both of these populations occurs at an E20 value of 0, with the frequency of 66,199 (20.7% of residues have an E20 value of 0). The maximum frequency of the second cluster appears to have an E20 value of 1 with 4.9% (15,576) of residues.

Figure 3.5B shows the frequency distribution of E6 for 319,551 query residues from the 1363 training data set. The E6 plot also displays similar shape and bimodal distribution as the 268 training set.

The FSR distribution plot as a function of frequency per query residue (Figure 3.5C) does not display patterns similar to those of the E20 and E6 distribution plots. The mode for FSR occurs at 0.0 with 51% of the residues at frequency of 162,822. Residues with an FSR value of 0 indicate an absence of small residues, whereas an FSR value of 1 indicates a high number of FSR (A and G). The FSHP distribution plot in Figure 3.5D also shows FSHP mode at zero, with 40% of the residues (126,484) displaying an FSHP value of 0. An FSHP value of 0 indicates an absence of substituted strongly hydrophobic residues.

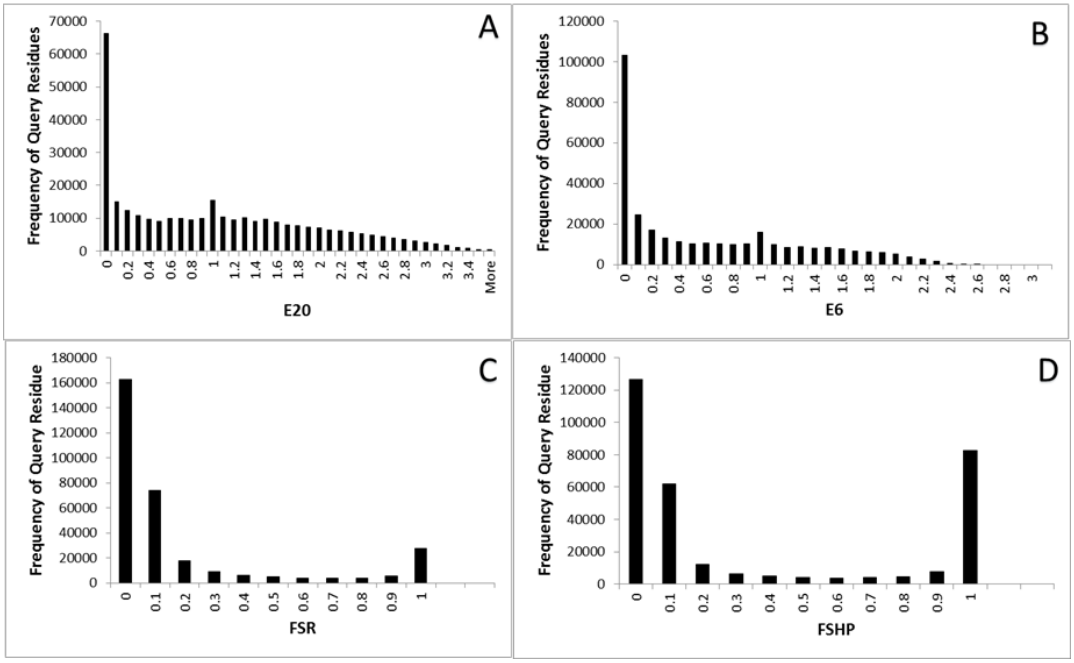

Figure 3.5. Frequency distributions for entropies and homology-based parameters of the 1363 learning set list of proteins. The 1363 proteins in the list have a total of 319,551 query residues, and total of 1,055,920 aligned subject protein sequences were used for these calculations. A. Frequency of query residues with respect to entropy values of the 1363 learning set. B. Frequency of query residues with respect to E6 values of the 1363 learning set. C. Frequency of query residues with respect to fraction small residues (FSR). D. Frequency of query residues with respect to fraction strongly hydrophobic residues (FSHP).

Similar to Figure 3.4, Figure 3.6 presents the frequency distribution of entropies and homology-based parameters, but for the 1363 training set for the two major regions, Region I and Region II. Figure 3.6A shows the E20 distribution for the two major regions, and 3.6B shows is the E6 distributions for the two regions. Similar to the E20, E6, FSR, and FSHP two-region distributions observed for the 268 training set (Figure 3.4), Figure 3.6 also presents similar trends and distribution plots.

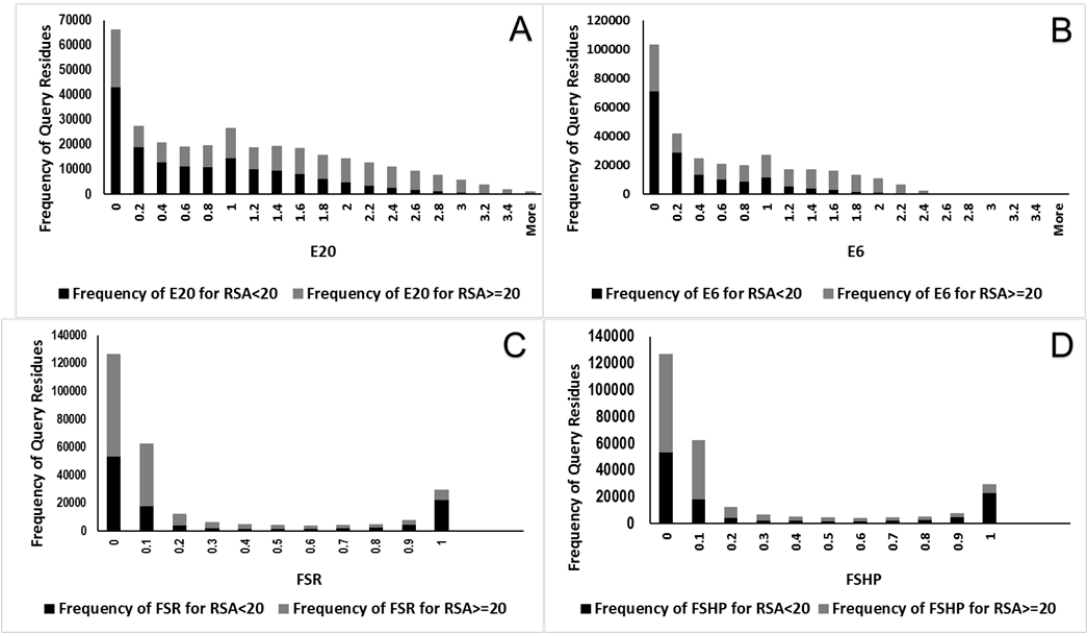

Figure 3.6. Frequency distributions of entropies and homology-based parameters of the 1363 training data set for the two major regions, Region I and Region II. A total of 1,055,920 residues were divided into Major Region I (RSA < 20) and Major Region II (RSA ≥ 20). A. E20; B. E6; C. FSR; D. FSHP.

Figure 3.7 presents the frequency distribution of NACCESS RSA values for 73,734 query residues of the 268 training data set. The mode of the distribution is at a NACCESS RSA value of 10, with 28.2% of residues (20,815) displaying this value. Any residue with a NACCESS ≥ 20 is characterized as being on the surface, whereas residues with a NACCESS value <20 are considered buried. Here in this distribution, the mode is 10, which indicates that a significant number of buried residues have a NACCESS value of 10. There are 28,256 residues with a NACCESS RSA of <20, indicating that 38.3% of

residues in the 268 training set are buried by X-ray. The remaining 61.7% of residues with NACCESS RSA values ≥ 20 are branded to appear on the surface.

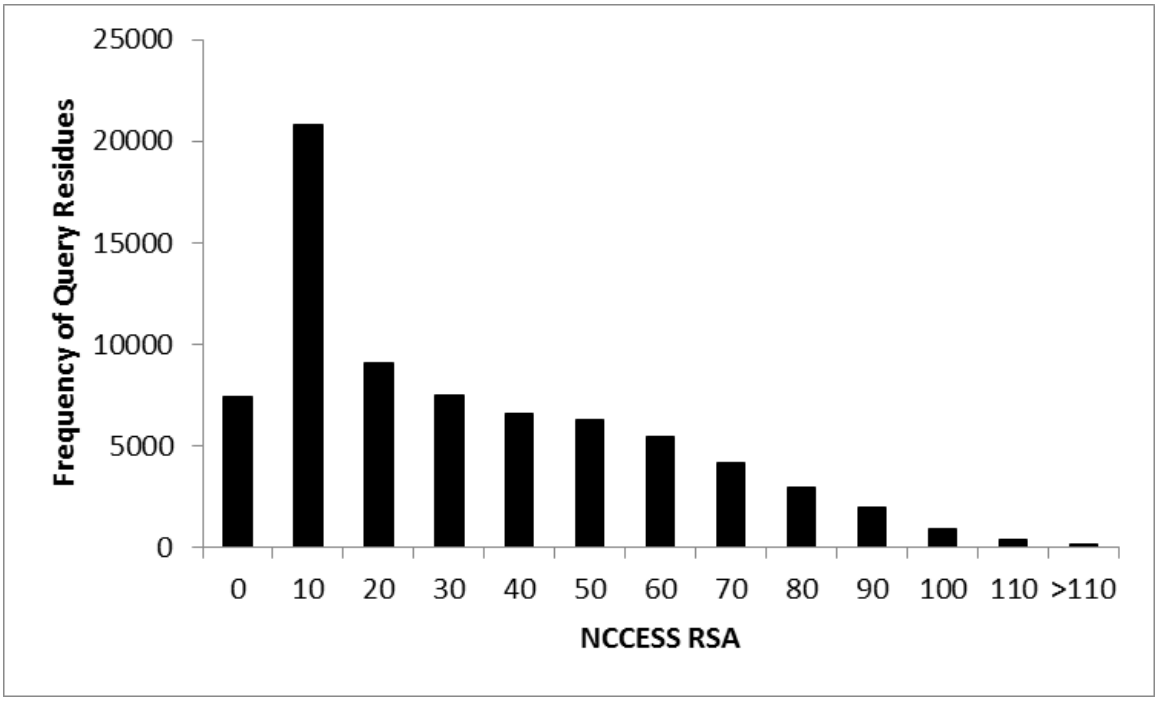

Figure 3.7. Frequency distribution of NACCESS RSA values for various RSA ranges for the 268 training data set. RSA values for a total of 73734 query residues were used for this plot.

Similarly to Figure 3.7, Figure 3.8 shows the frequency distribution of NACCESS RSA values, but for 319,812 residues of the 1363 training data set. In comparison to the 268 NACCESS RSA distribution plot, the 1363 training set also displays a similar trend. The mode of distribution for the 1363 training set also occurs at a NACCESS RSA value of 10. The 1363 training set also has 38.3% of residues characterized as buried and 61.7% of residues on the surface. The mean, mode, and median of this distribution are 27.4, 0, and 19.4, respectively.

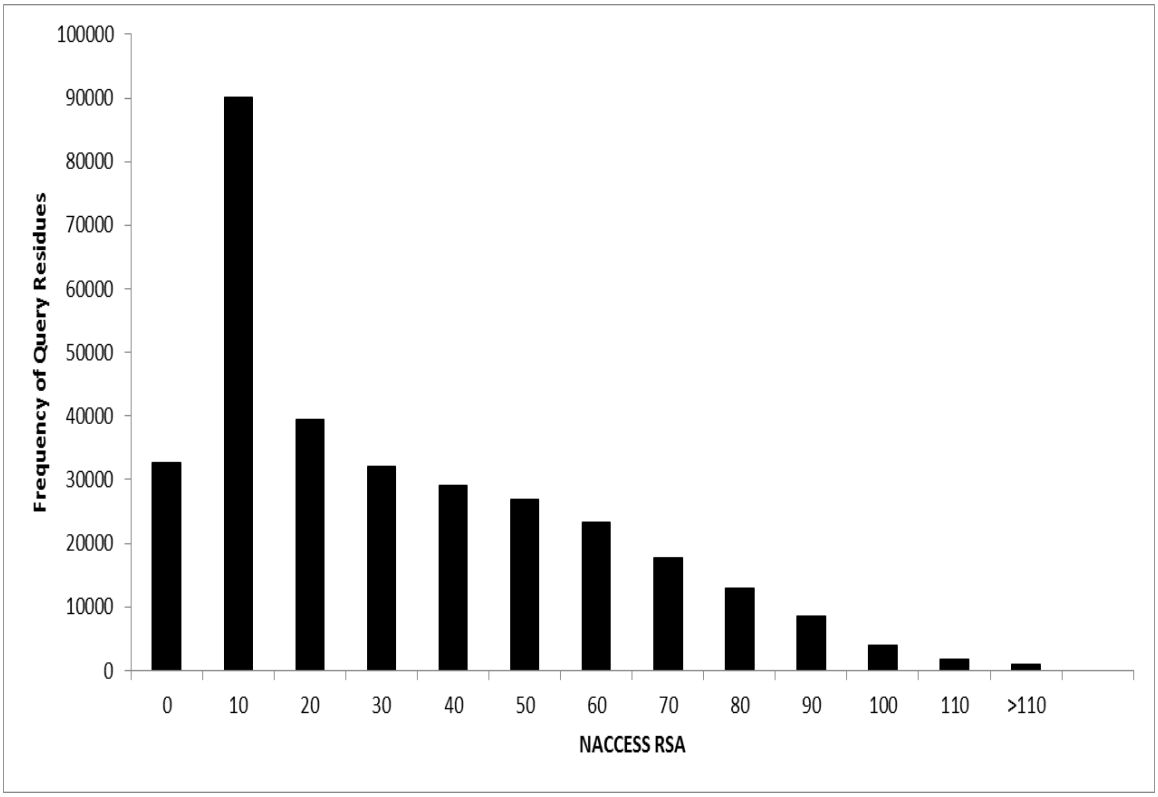

Figure 3.8. Frequency distribution of RSA values for various RSA ranges for the 1363 training data set. RSA values for a total of 319,812 query residues were used for this plot.

Figure 3.9 represents a comparison of RSA distribution of 50,856 residues of the 215 test set for NACCESS RSA and predicted RSA values generated using the 268 training set. The NACCESS RSA value distribution plot is displayed in Figure 3.9. The NACCESS frequency distribution peaks at a NACCESS value at 5, with a total of 9572 residues (18.8%) displaying this NACCESS value. Both distributions are right-skewed histograms. The NACCESS threshold for buried resides is ≥ 20, and this distribution indicates about 48.4% (24,595) of residues can be characterized as buried when derived from X-ray information. The remaining 51.6% of residues are characterized as appearing on the surface of the protein. The mean, median, and mode of this distribution are 28.5,

21.6, and 0, respectively.

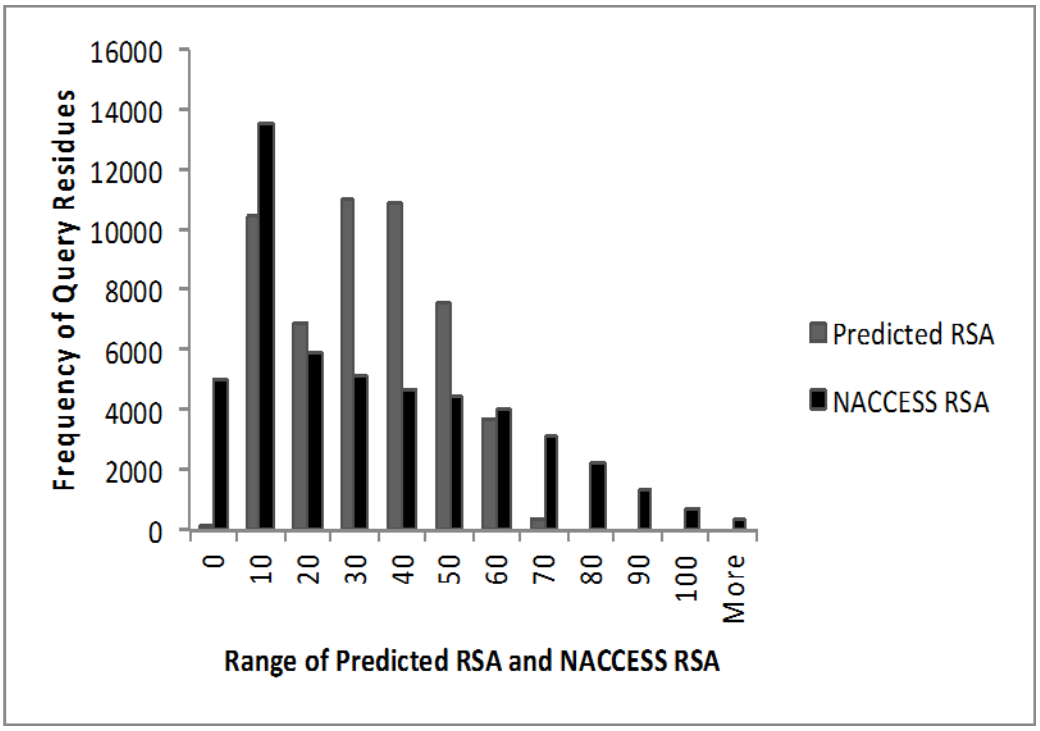

Figure 3.9. Frequency distribution comparison of NACCESS RSA values and predicted RSA values for the 215 test set using the 268 training set. A total of 50,856 residues for a total of 215 protein lists were used to generate these plots. Frequency of query residues with respect to NACCESS RSA values and frequency of query residues with respect to predicted RSA values generated by linear regression are presented.

The second part of Figure 3.9 shows the distribution of predicted RSA values for the 50,856 residues of the 215 test set. Unlike the NACCESS plot, the predicted RSA plot peaks at an RSA value of 10 with a total of 9729 (19.1%), and the second highest peak is observed at a predicted RSA of 35 with 5097 (10.0%) residues with an RSA of 35. The mean of this distribution is 27.1, the median is 27.4, and the mode is 6.4. Unlike NACCESS RSA, where there are 4767 residues (9.4%) with NACCESS values greater than 70, none of the residues in the predicted RSA are forecasted to have RSA value

greater than 70. Also, predicted RSA values appear compressed relative to NACCESS RSA.

The comparison of NACCESS RSA and predicted RSA for the 215 test set using 1363 data as a training set is presented in Figure 3.10. Just as in first part of Figure 3.9, Figure 3.10 presents NACCESS RSA distribution for the 50,856 residues of the 215 test set. The second part of Figure 3.10 is the distribution of predicted RSA values for the 215 test set using the 1363 as training set. The highest peak in the predicted RSA generated via using the 1363 as training set is observed at an RSA value of 10. A total of 8693 residues (17.1%) have predicted RSA values equal to 10. Similar to the distribution plot of the 215 test set using the 268 as training set (Figure 3.9), very few residues (0.1%) are predicted to have RSA value equal to 5, the highest corresponding peak in the NACCESS RSA frequency. Again, none of the residues are predicted to have RSA values greater than 70. The mean, median, and mode of this distribution are 28.1, 28.7, and 7.6, respectively. This distribution displays a multimodal distribution pattern.

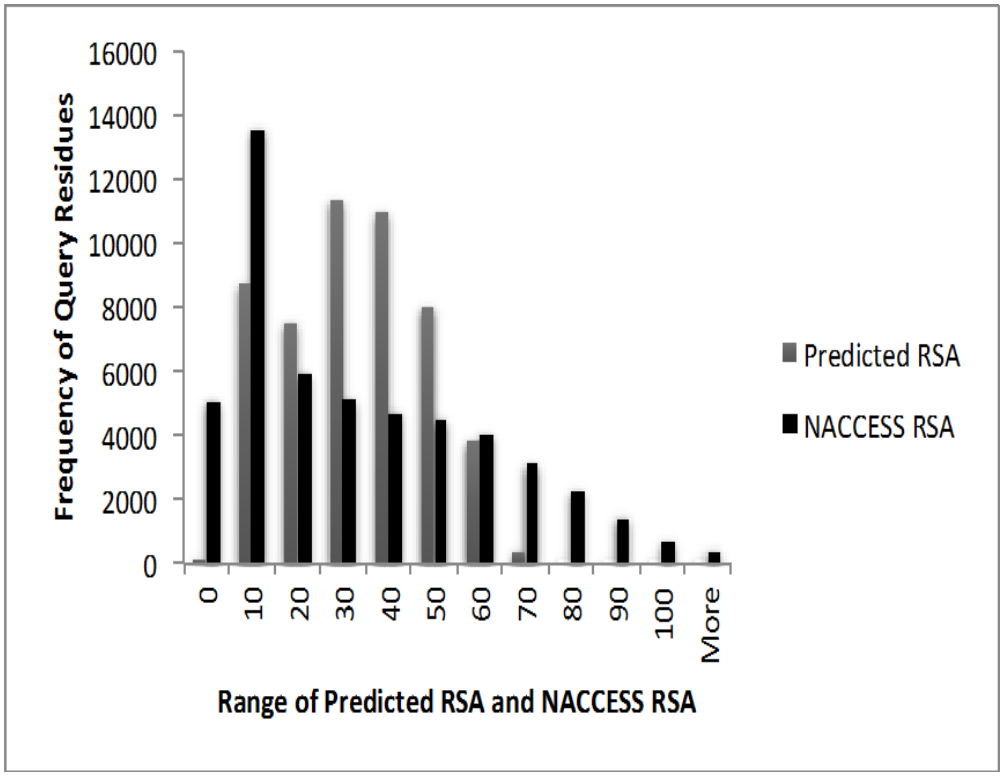

Figure 3.10. Frequency distribution comparison of NACCESS RSA values and predicted RSA values for the 215 test set using the 1363 training set. A total of 50,856 residues for 215 protein lists were used to generate these plots. Frequency of query residues with respect to NACCESS RSA values and frequency of query residues with respect to predicted RSA values generated by linear regression are displayed.

Figure 3.11 presents comparison aggregate correlation plots of sequence entropy and other homology-based parameters for the 268 training data set. Here is the combined aggregate correlation plot for the manually generated BLAST output calculation performed previously by the Lustig group (Mishra, 2010). On the other hand, Figure 3.11B is the combined aggregate correlation plot for automatically generated BLAST output calculations. This comparison was made to validate the finding of the automated BLAST output files. Aggregate values for all of the homology-based parameters are determined by averaging their respective values within the same packing density interval.

The standard deviation for both the manually generated and automatic calculations are comparable, typically 0.3 for E20 and E6, and 0.1 for FSHP and FSR.

Comparable to our previous work (Mishra, 2010; Rose et al., 2011), two major regions were noted in in both the graphs. Major Region I is associated with a packing density of 11 to 25 (0.09 to 0.04 of inverse density), and relates to the portion of the graph where average sequence entropy increases linearly with an increase in packing density. On the other hand, in Major Region II, associated with a packing density of 4 to10 (0.25 to 0.1 of inverse density), a different trend is observed. In Major Region II, sequence entropy remains almost the same as packing density increases. Both Figures 3.11A and 3.11B show similar trends and patterns.

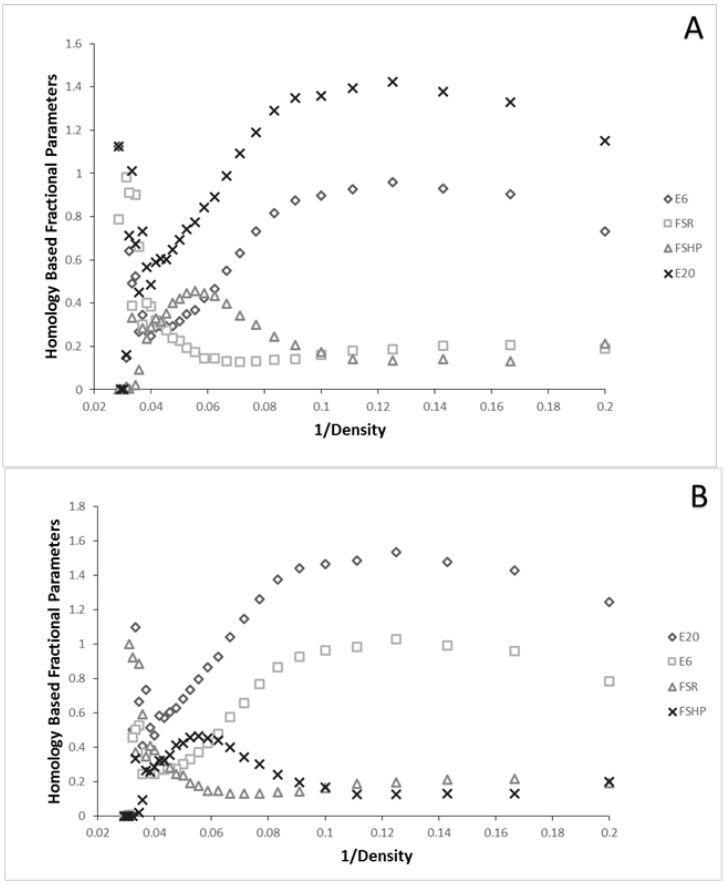

Figure 3.11. Comparison of combined aggregate correlation plots of sequence entropy and other homology-based parameters for the 268 training set. Packing density is the number of Cα within a 9Å radius, and excluded here is the portion of Region II with packing densities less than 5 (<1% of all residues). Average sequence entropy, E20 (open-square, ordinate) and E6 (closed-diamond), are calculated by averaging the respective values for 73,727 query residues for each inverse packing density value. Fraction of strongly hydrophobic residues (asterisk) and fraction of small residues (open-diamond) are calculated and averaged over a total of 7.12E7 aligned residues, plotted against inverse packing density. Average values for all the homology-based parameters are determined by averaging their respective values within the same packing density interval. Note that the standard deviations for E20 and E6 are comparable (typically 0.3), while typically 0.1 for FSHP and FSR. A. Combined aggregate correlation plot for the manually generated BLAST output calculation (Mishra, 2010). B. Combined aggregate correlation plot for the automatically generated BLAST output calculation.

Figure 3.12 shows the aggregate correlation plots of sequence entropy and other homology based parameters for the 1363 training data set. The presence of the two major Regions I and II are also observed with comparable trends in the 1363 training set as with the 268 training data sets.

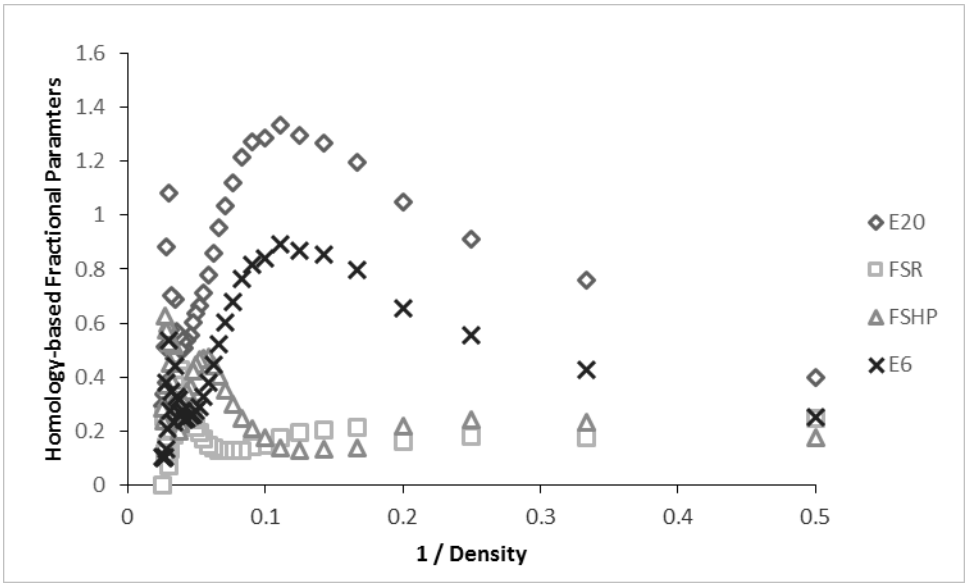

Figure 3.12. Combined aggregate correlation plots of sequence entropy and other homology-based parameters for the 1363 training set. Packing density is the number of Cα within a 9Å radius and that the portion of Region II with a packing density less than 5 is <1% of all residues. Average sequence entropy and E6 are calculated by averaging the respective values for 73,734 query residues for each inverse packing density value. Fraction of strongly hydrophobic residues and fraction of small residues are calculated and averaged over a total of 318,840 aligned residues, plotted against inverse packing density. Average values for all the homology-based parameters are determined by averaging their respective values within the same packing density interval. Note that the standard deviations for E20 and E6 are comparable (typically 0.3), while typically 0.1 for FSHP and FSR.

Figures 3.13 and 3.14 represent individual plots of RSA as a function of inverse packing density and packing density for the 268 and 1363 training data sets. Again,

similar to the aggregate correlation plots of sequence entropy, two major regions are observed with aggregate plots of RSA. In the first major region, RSA values decrease linearly as packing density increases, and in the second major region, RSA values stay almost constant—close to 0—as packing density increases. Similar trends are observed in both of the training data sets. These concur with our previous findings (Mishra, 2010) that residues associated with lower packing densities have higher RSA and are more likely to be found on the surface of the proteins, whereas residues that have a high packing density are dense and are usually found in the core of the proteins. The RSA values for these residues should be closer to 0. Figures 3.12 and 3.13 support this claim, as high-density values have RSA close to 0. Although majority of protein residues display this trend, very few residues have RSA values of 0 for a low packing density. Figures 3.13A and 3.14A simply show this RSA trend as a function of inverse density.

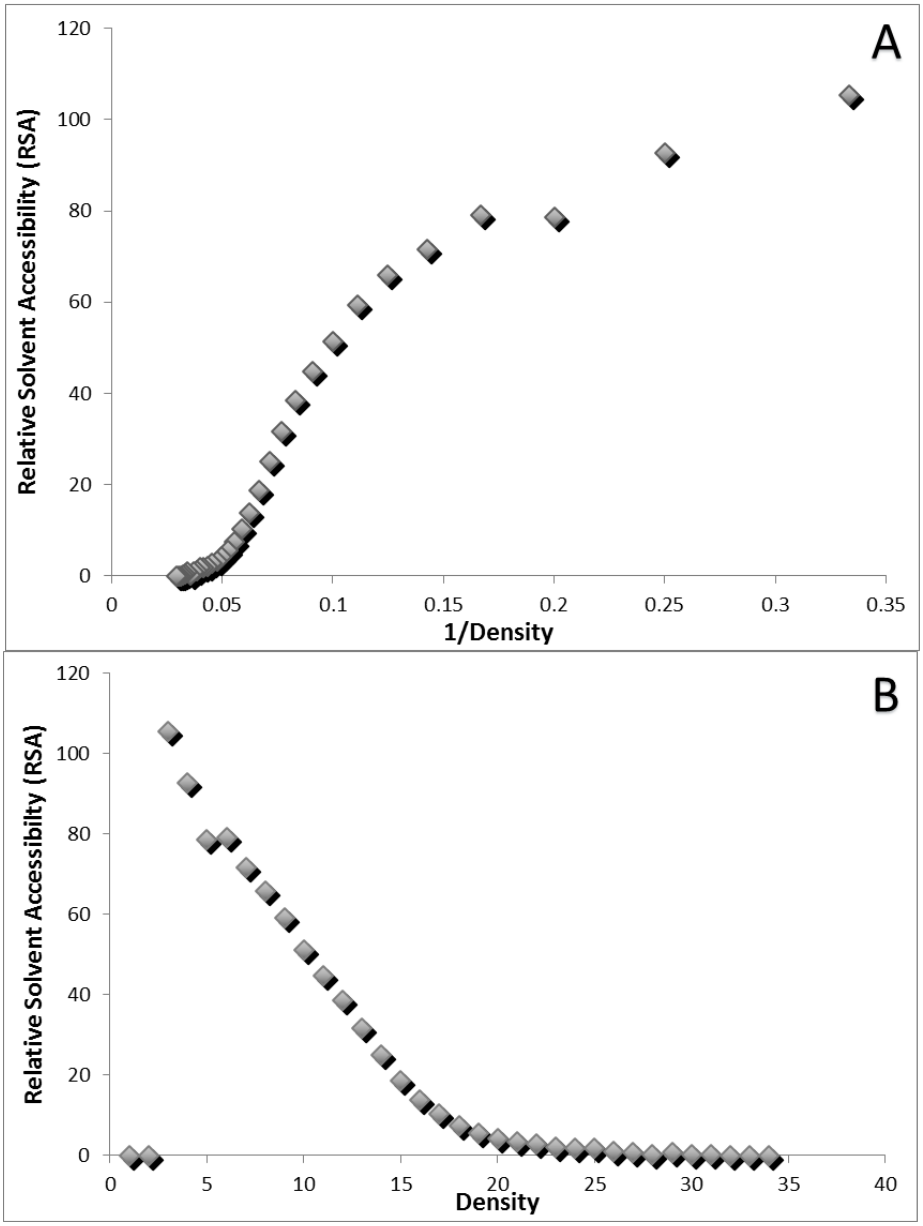

Figure 3.13. Density—relative surface accessibility comparison for the 268 training set. Here the aggregate of RSA values were obtained by averaging a total of 73,734 query residues at each packing density position. A. Aggregate correlation plot of relative surface accessibility (RSA) and inverse packing density for the 268 training set of proteins. B. Aggregate correlation plot of relative surface accessibility (RSA) and packing density for the 268 training set of proteins.

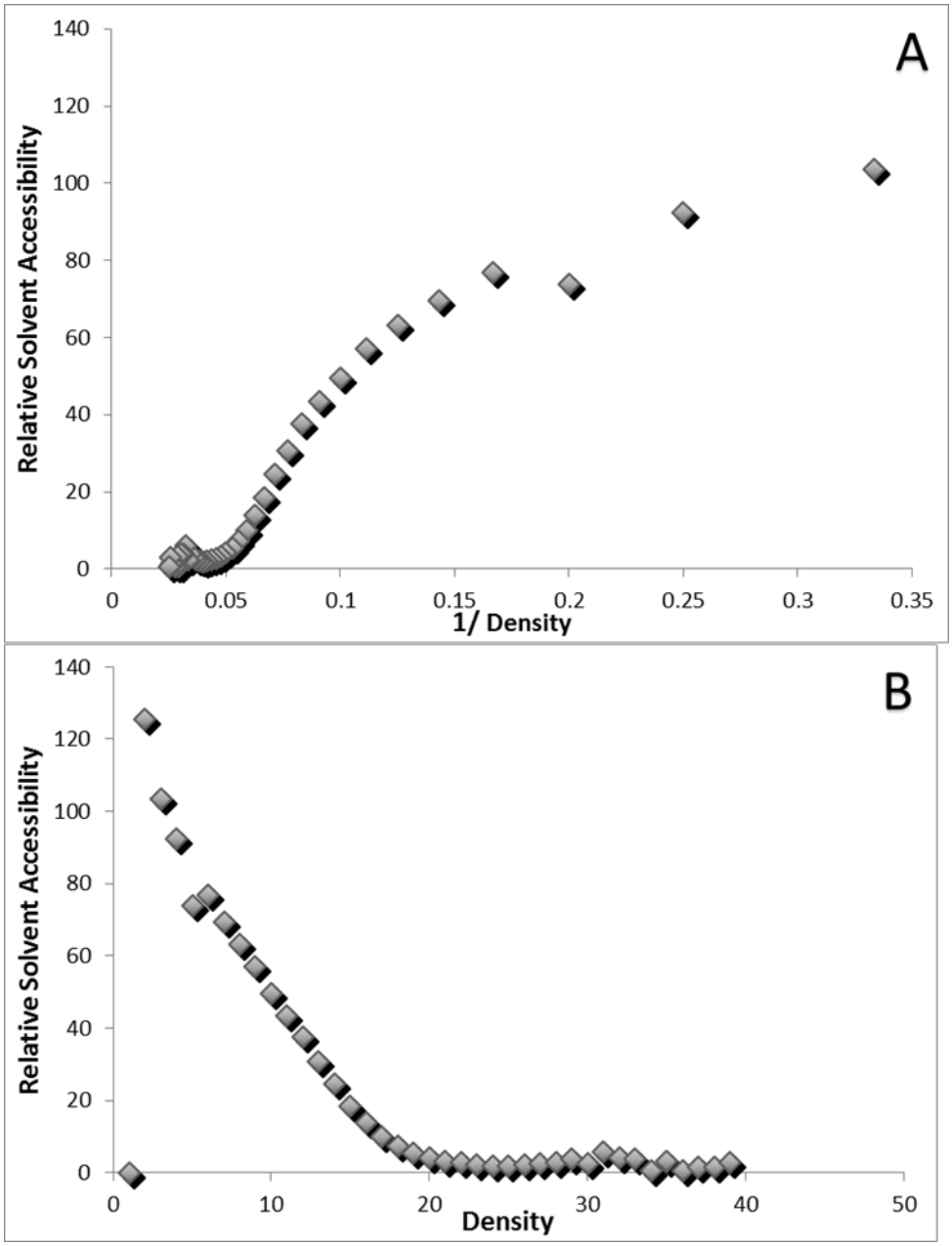

Figure 3.14. Density—relative surface accessibility comparison for the 1363 training set. Here the aggregate of RSA values was obtained by averaging a total of 318,840 query residues at each packing density position. A. Aggregate correlation plot of relative surface accessibility (RSA) and inverse packing density for the 1363 training set of proteins. B. Aggregate correlation plot of relative surface accessibility (RSA) and packing density for the 1363 training set of proteins.

3.2 Accuracy of Results

Table 3.1 showcases a comparison of accuracies for the 12 different linear regression models using the 268 training data set on the 215 test data set for the manually derived BLAST and automated BLAST outputs. The accuracy for the manually derived BLAST output is generally lower than the accuracy of its counterpart model using the automated BLAST output. The accuracies were generated with the same number; 73,734 aligned residues were used for the 268 training data set, and 50,856 aligned residues were used for the 215 test set. Theoretically, these two sets of accuracies should be identical, but due to various parameters involved during the process of accuracy calculation they are not. Previous R scripts (Rose et al., 2011) calculated the manual BLAST output accuracies. First, linear regression for the training data set was generated, and then β coefficients that were to be applied to the test set were extracted. The β coefficients were also manually fed into the 215 data set RSA prediction calculations. This manual introduction of β coefficients into the code presented room for error in calculation. Anytime there was an improvement or modification made to the training set, it altered the β coefficients' values. Due to the fact that the entries were made manually, these changes were often not reflected in the application to the test set. The possibility of input truncation was also introduced.

However, in the automated BLAST output accuracies, this sort of error in calculations was eliminated by sequentially generating the linear model using the training data set and applying the parameters obtained from the regression to the test sets automatically and internally in the program. Anytime there were any changes implied in

the training set, those changes were automatically reflected in the test set. This new automated accuracy code was also implemented in the manual BLAST output calculations, and comparable accuracies with the automated BLAST outputs were observed. The highest accuracy observed was for the E20 + E6 + FSR + FSHP + AA model at 72.4% and 74.2%, respectively, for the manual and automated version of the 268 training set and the 215 test set. Accuracy of 74.4% was also observed for the E6 + FSR + FSHP+ AA model. Noteworthy is the fact that the E20 + E6 + AA hovers very close to the highest accuracy at 74%. This also shows that the addition of a qualitative predictor, AA, aided in better accuracy prediction because, without AA, the E20 + E6 model has accuracy of 69.1%, which is lower by 4.9%. Also, the model FSHP + AA is missing from the manual BLAST accuracy because FSHP + AA was incorporated as the 12^{th} model after these values were generated.

Table 3.1. Comparison of regression accuracy using manually generated BLAST output calculation and automatically generated BLAST output calculations.

Models	**Accuracy**	
	Manual 268 Training 215 Test	Automated 268 Training 215 Test
E20	63.1	63.0
E6	67.0	68.9
FSHP	67.3	68.2
FSHP + AA		69.8
AA	70.6	70.4
E20 + AA	72.1	72.8
E6 + AA	73.1	74.0
E20 + E6+ AA	72.9	74.0
E20 + E6	67.4	69.1
E20 + FSR + FSHP + AA	73.4	73.3
E6 + FSR + FSHP + AA	73.2	74.2
E20 + E6 + FSR + FSHP + AA	72.4	74.2

Table 3.2 displays a summary of regression accuracies for 12 models tested for the 215 test set using 268 as the training set, the 215 test set using 1363 as the training set, and finally the 215 PSI-BLAST test set using 268 PSI-BLAST as the training set. This set of 12 models accuracy calculations is also referred to as standard accuracy calculation for the remainder of this thesis. The NACCESS RSA threshold used for the 268 training set, and the 268 PSI-BLAST training set was >23, while >25.2 was used for the 1363 training set. These thresholds were determined with the aid of linear regression line of best fit (Figures 2.2 and 2.3) for each of the training sets.

Here again as in Table 3.1, we observe similar trends. Consistently, for all of the training and test set combinations used, E20 + E6 + FSR + FSHP + AA has the highest prediction accuracy observed. Our findings here indicate that, although the highest

accuracy for the PSI-BLAST model does slightly better, at 74.4% vs. 74.2% for regular BLAST, it does not have a significantly large impact. Also for some models, like E6 + FSR + FSHP + AA, PSI-BLAST actually results in lower accuracy than BLAST at 69.09% vs. 73.29%. As was seen in Table 3.1, the addition of the qualitative predictor AA improves accuracy.

Table 3.2. Summary of regression accuracy for the 12 models tested. The three different sets of numbers represent different training and test set models. The predicted RSA thresholds for the three sets were slightly different depending on the non-hydrophobic linear regression intercept for each data set.

Models	**Accuracy**		
	268 Training 215 Test[1]	1363 Training 215 Test[2]	268 PSI-BLAST Training 215 PSI-BLAST Test[3]
E20	63.0	63.3	63.1
E6	68.9	68.8	68.8
FSHP	68.3	68.2	68.2
FSHP + AA	69.7	70.8	70.8
AA	70.4	70.4	70.4
E20 + AA	72.8	72.9	72.9
E6 + AA	74.0	74.1	74.1
E20 + E6+ AA	74.0	74.1	74.1
E20 + E6	69.1	69.1	74.1
E20 + FSR + FSHP + AA	73.3	74.1	69.1
E6 + FSR + FSHP + AA	74.2	74.4	74.1
E20 + E6 + FSR + FSHP + AA	74.2	74.4	74.4

[1] Predicted RSA threshold used >23.
[2] Predicted RSA threshold used >25.2.
[3] Predicted RSA threshold used >23.

3.3 Outcome of Additional Methods to Improve Prediction Accuracy

3.3.1 Outcome of Additional Models Applied

Separate predictors for each of the small residues Ala and Gly are worth noting here. In our previous models, Ala and Gly were incorporated under one sequence homology parameter, FSR. The fraction of aligned residues that are Gly (FG) and Ala (FA) were then generated. New models that substituted FSR with FA, FG, and FA + FG were generated. The accuracies for each of these additional models are presented in Table 3.3. The highest accuracy observed as a result of the incorporation of these additional models was 73.6% and 73.4% for the 268 and 1363 training data sets, respectively. Contrary to our hypothesis, separating the Ala and Gly from the existing FSR sequence homology did not improve the overall prediction accuracy. Interestingly for both the 268 and 1363 training sets, the accuracies with the 215 set hovered between 73.0% and 73.6 %. While we observed variant accuracy ranges for all of the other regression models we applied (see Tables 3.1 and 3.2), it is noteworthy that the range of accuracies involving FG and FA did not improve over the standard methods.

Table 3.3. Regression accuracy table for additional models applied. The two different sets of numbers represent different training and test set models. The predicted RSA thresholds for the two sets were slightly different depending on the non-hydrophobic linear regression intercept for each data set. >23 and >25.2 were used, respectively, for the 268 BLAST–215 BLAST and1363 BLAST–215 BLAST.

Models	Accuracy	
	268 Training 215 Test	1363 Training 215 Test
FSHP + AA	69.7	70.8
E20 + FA + FG + FSHP + AA	73.3	73.0
E20 + FA + FSHP + AA	73.3	73.0
E20 + FG + FSHP + AA	73.2	73.0
E6 + FA + FG + FSHP + AA	73.6	73.4
E6 + FA + FSHP + AA	73.6	73.4
E6 + FG + FSHP + AA	73.4	73.4
E20 + E6 + FA + FG + FSHP + AA	73.6	73.4
E20 + E6 + FA + FSHP + AA	73.6	73.4
E20 + E6 + FA + FSHP + AA	73.5	73.4

3.3.2 Outcome of Categorized Protein Data Set

In the Hotpatch study by Petit and coworkers (2007), a total of 618 PDB IDs are included. Out of the two categories investigated to explore the possibility of accuracy improvement, only the protein binding group yielded better accuracy. Table 3.4 displays protein binding PDB ID matches between the 618 set and each of the two training data sets (268 and 1363) and the test set. There were 13 PDB ID matches between the 618 and 268 training sets, 8 PDB ID matches between the 618 and 1363 training sets, and a total of 16 PDB ID matches between the 618 and 215 test sets for the protein-binding category.

Table 3.4. Protein binding category PDB ID matches between the 618 Hotpatch protein PDB IDs (Petit et al., 2007) and the 268 and 1363 training sets and 215 test set.

	Protein Binding PDB ID Matches		
	618–268 matches	618–1363 matches	618–215 matches
1.	1AK4C	1BKRA	1BEOA
2.	1B3AA	1G3PA	1BGCA
3.	1BUOA	1KPTA	1CFYA
4.	1FINB	1KWAA	1CSGA
5.	1KPTA	1TENA	1JKWA
6.	1KWAA	1VCAA	1KNBA
7.	1M6PA	2PSPA	1KPTA
8.	1OSPO	3SEBA	1LKIA
9.	1YCSA		1LKKA
10.	1YCSB		1MAZA
11.	2ILKA		1OSPO
12.	2TGIA		1SIGA
13.	2TRCP		1SVPA
14.			1VCAA
15.			1WHIA
16.			2PSPA

There were two sets of regressions carried out for the protein binding categories. For the first set of regression analyses, 618-268 protein binding (PB) matches were used as the training set and 618-215 protein binding matches were used as the test set. For the second set of regression analyses, 618-1363 protein binding matches were used as the training set and 618-215 protein binding matches were used as the test set. The prediction accuracy for the 12 different models is presented in Table 3.5.

As observed from the accuracy table (see Table 3.5), the PB category yielded slightly better prediction accuracy than our previous regression analysis (see Tables 3.1, 3.2, and 3.4). In standard calculation, the highest accuracy achieved was 74.4% for the 1363 training set and the 215 test set calculations. The highest accuracy value for the PB

is at 76.3% for 618-268 PB matches as training set and 618-215 PB matches for the test set. Both E20 + E6 + FSR + FSHP + AA and E6 + FSR + FSHP + AA display the highest predicted accuracy for this regression analysis at 76.0. Both sets of regressions display similar patterns and accuracy amounts.

Table 3.5. Regression accuracy table for protein binding model. The two different sets of numbers represent different training and test set models. The predicted RSA thresholds for the two sets were >23 and >25.2, respectively, for the 618-268 and 1363-268 matches. Here, PB stands for protein-binding.

Models	**Training = 618-268 PB PDB IDs Matches Test = 618-215 PB PDB IDs Matches[1]**	**Training = 618-1363 PB PDB IDs Matches Test = 618-215 PB PDB IDs Matches[2]**
E20	60.5	60.5
E6	60.5	60.5
FSHP	73.0	74.3
FSHP + AA	74.2	75.8
AA	73.0	73.4
E20 + AA	74.8	74.5
E6 + AA	75.4	75.4
E20 + E6 + AA	75.0	75.5
E20 + E6	60.1	61.4
E20 + FSR + FSHP + AA	76.0	75.3
E6 + FSR + FSHP + AA	76.2	76.0
E20 + E6 + FSR + FSHP + AA	76.3	76.0

[1] The predicted RSA threshold used for this set of calculations was >23.
[2] The predicted RSA threshold used for this set of calculations was >25.2.

3.3.3 Use of All 618 PDB IDs as Training and 215 as Test

Another test performed on the 618 set was the use of all 618 proteins as the training set and the 215 as the test set. There were four different sets of linear regression calculations involving the 618 and 215 protein sets: 1.generic, 2. protein binding,

3. Oligomers, 4. generic without PB and Oligomers. Table 3.6 includes a comprehensive presentation of the actual application of these sets of calculations.

Table 3.6. Description of 618 PDB IDs as training and 215 PDB IDs as test regression analysis calculations. This table summarizes the different groups of PDB IDs used as training and test sets for this analysis.

	Training	**Test**
1) Generic	All of the 618 protein	All the PDBs in the 215 set that match 618 set
2) Protein Binding (PB)	PB PDB IDs from 618 Set[1]	PB PDBs matches between the 618 and 215 sets
3) Oligomers	Oligomer PDB IDs from the 618[2]	Oligomer PDBs matches between the 618 and 215 sets
4) Generic without PB and Oligomers	618 PDBs excluding PB and oligomers[3]	All 618 – 215 matches without PB and oligomers

[1]618–215 PB matches were excluded.
[2]618–215 Oligomer matches were excluded.
[3]618–215 matches were excluded.

Table 3.7 is the regression accuracy results of the different set calculations presented in Table 3.6. Out of the four categories, protein binding consistently had higher accuracies for each of 12 models tested, while oligomers reliably had lower accuracies for each model. Protein binding model, RSA = E20 + FSR + FSHP + AA, had the utmost accuracy, 77.7%, of all the models for the various categories tested. The addition of the amino acid (AA) qualitative predictors to each of the homology-based parameters used as models generally resulted in higher accuracies compared to the same model without an AA qualitative predictor.

Table 3.7. Regression accuracy table of PDB IDs of the four groups: 1. generic 2. protein binding 3. oligomers 4. and generic without PB and Oligomers used in the regression analysis. Here the entire categorized 618-protein sets were used as training, and

Model	Generic	Protein Binding (PB)	Oligomer	Generic without PB and Oligomer
RSA = E20	62.1	60.5	61.0	63.3
RSA = E6	68.8	60.5	68.2	70.1
RSA = FSHP	69.8	74.0	68.3	71.9
RSA = FSHP + AA	71.0	75.8	69.8	72.3
RSA = AA	71.2	74.4	70.3	70.9
RSA = E20 + AA	73.5	76.0	72.3	74.6
RSA = E6 + AA	74.6	76.9	73.1	75.1
RSA = E20 + E6 + AA	74.5	76.7	73.2	75.2
RSA = E20 + E6	69.1	63.1	68.7	70.3
RSA = E20 + FSR + FSHP + AA	74.0	76.9	72.7	75.1
RSA = E6+ FSR + FSHP + AA	74.7	77.2	73.5	75.3
RSA = E20 + E6 + FSR + FSHP + AA	74.7	77.3	73.4	75.4

categorized PDB IDs from the 215 were used as test set.

3.4 Incorporation of the Categories to the Existing Data Sets

There are several PDB IDs that match PDB IDs in the 268, 215, and 1363 data sets. The last set of regression analyses took advantage of the PDB IDs that overlap between the different data sets with 618. Similar to the previous set of calculations with

the 618 data set, these regression calculations were also calculated for four categories of PDB ID: 1. Generic, 2. protein binding (PB), 3. Oligomers, 4. generic without PB and oligomers.

As the first task, all the PDB IDs that were common between the 618 and 215 data set, 618 and 268 data set, and 618 and 1363 data set were extracted for each of the categories. The protein binding category had a total of 17 PDB IDs in 215, 12 in 268, and 8 in 1363 common with the 618 protein binding category. The oligomer category shared 30 PDB IDs in 215, 25 in 268, and 12 in 1363 with 618 in the oligomer category. Generic PDB IDs are all of the PDB IDs in the various categories in the 618 data set. Generic matches between each of the data sets simply are the number of PDB IDs common between each of the data sets with the 618 data set. All PDB ID matches between the 618 data set and each of the three data sets (268, 215, and 1363) are presented in Tables 3.8 through 3.10.

Tables 3.11 through 3.14 are the regression accuracies for the different categories using PDB IDs that are common between the 618 data set and each of the three original data sets (268, 1363, and 215). Table 3.11 is the regression accuracy for the generic category. Generic PDBs are the PDBs that were found in common between the 618 data set and the 268, 215, and 1363 group for all of the categories listed in the 618 data set. The highest accuracy for the generic category was achieved at 74.8% for the RSA = E20 + E6 + FSR + FSHP + AA model using all of the 618-268 common PDBs as the training set and all of the 618-215 common PDBs as the test set.

The regression accuracies for the 12 models for the protein binding category are presented in presented in Table 3.12. The highest accuracy for the protein binding category is 76.9% for the RSA = E20 + E6 + FSR + FSHP + AA model using protein binding from 1363 as the training set and protein binding from 215 as the test set. Similar to earlier calculations, the protein binding category yielded the overall highest prediction accuracies.

Table 3.8. PDB ID matches between the 618 data set with three data sets (215, 268, and 1363) for the protein binding and oligomer categories. Note: A PDB ID categorized as protein binding can also be categorized as oligomer.

Protein Binding			Oligomers		
215	268	1363	215	268	1363
1BEOA	1YCSB	1BKRA	1QAPA	1KPFA	2PSPA
1BGCA	1OSPO	2PSPA[1]	1DOSA	3CLAA	1A73A
1CFYA	1AK4C	3SEBA	1HGXA	1AJSA	1HFES
1CSGA	1BUOA	1KPTA	2PSPA[1]	2SICI	1UTGA
1DKTB	1B3AA	1TENA	1SVPA[1]	1BD0A	1MTYB
1JKWA	2TGIA	1KWAA	1BBPA	1BUOA	1B5EA
1KNBA[1]	1KPTA	1VCAA	1TFEA	1VLBA	3CHBD
1KPTA	1FINB	1G3PA	1BTMA	1AORA	1PSRA
1LKIA	1M6PA		1GSAA	1B3AA	1BEBA
1LKKA	1KWAA		1KNYA	1UTGA	1REGX
1MAZA	2ILKA		3SDHA	2TGIA	2SQCA
1OSPO	2TRCP		1ABRB	3SDHA	1OTFA
1SIGA			1IDAA	1B5EA	
1SVPA[1]			1ECPA	1GOTB	
1VCAA			1DELA	1GVPA	
1WHIA			3MINB	1CG2A	
2PSPA[1]			1GOTB	1NOXA	
			1PDOA	1M6PA	
			1DKZA	3DAPA	
			1NOXA	2ILKA	
			1PEAA	12ASA	
			2TYSA	1REGX	
			1FDSA	1HJRA	
			1XVAA	1DPGA	
			1AFRA	2SQCA	
			1HSBA		
			1OFGA		
			1YASA		
			1DHRA		
			1KNBA[1]		

[1] Represents PDB IDs that is common between the protein binding category and Oligomer Category.

Table 3.9. PDB ID matches between the 618 data set with three data sets (215, 268, 1363) between all categories.

Generic						
215			268		1363	
1LKKA	3SDHA	1ABRB	1YCSB	1KWAA	1BKRA	2PIAA
1MAIA	5P21A	1AFRA	1KPFA	1AYLA	1XFFA	1REGX
1MAZA	1KTEA	1AXNA	3CLAA	1THTA	1PINA	1G3PA
1NOXA	1LBAA	1BBPA	1FJMA	2ILKA	2PSPA	2SQCA
1OFGA	1LCLA	1BEOA	1OSPO	12ASA	2SCPA	1OTFA
1OSPO	1LKIA	1BGCA	1AK4C	2TRCP	256BA	
1PDOA	2PSPA	1BIBA	2SCPA	1REGX	1A73A	
1PEAA	2RN2A	1BTMA	1AJSA	1NO3A	1IXHA	
1POCA	2SCPA	1BTNA	2SICI	1HJRA	3SEBA	
1POTA	2TYSA	1CFYA	1TX4A	1DPGA	1UTGA	
1QAPA	3CHYA	1CHDA	256BA	1STFI	2FDNA	
1RCFA	3MINB	1CNVA	1NP4A	2SQCA	1A62A	
1RECA	1IDAA	1CSGA	2LIVA	2MBRA	1MTYB	
1RSYA	1IDOA	1DELA	1BD0A	1A48A	1B5EA	
1SBPA	1JKWA	1DHRA	1BUOA	1A6QA	1KPTA	
1SIGA	1KNBA	1DKTB	1VLBA	1GVPA	1AMUA	
1SMEA	1KNYA	1DKZA	1HIAI	2RN2A	1MPGA	
1SRAA	1KPTA	1DOSA	1AORA	1CG2A	1IIBA	
1STFI		1ECEA	1BLZA	1NOXA	1FDRA	
1SVPA		1ECPA	4HTCI	1FINB	3CHBD	
1TFRA		1EXNB	1B3AA	1M6PA	1NBCA	
1VCAA		1FDSA	1UTGA	3DAPA	1CIPA	
1WHIA		1FJMA	2TGIA	2TPSA	1TENA	
1XVAA		1FTPA	1AK0A		2TPSA	
1YASA		1GAIA	3SDHA		1KWAA	
256BA		1XFFA	1B5EA		1JFRA	
2AYHA		1GPCA	1AH7A		1PSRA	
2GDMA		1GSAA	13PKA		1VCAA	
2LIVA		1HGXA	1KPTA		1A8LA	
2MTAC		1HLBA	1AMUA		1BEBA	
2PIAA		1HSBA	1MPGA		2GDMA	

Table 3.10. PDB ID matches between the 618 data set and the three data sets (215, 268, and 1363) between all categories without the protein binding and oligomer matches.

Generic without Protein Binding and Oligomers			
215		268	1363
1AXNA	1SBPA	1FJMA	
1BIBA	1SMEA	2SCPA	1XFFA
1BTNA	1SRAA	1TX4A	1PINA
1CHDA	1STFI	256BA	2SCPA
1CNVA	1TFRA	1NP4A	256BA
1ECEA	256BA	2LIVA	1IXHA
1EXNB	2AYHA	1HIAI	2FDNA
1FJMA	2GDMA	1BLZA	1A62A
1FTPA	2LIVA	4HTCI	1AMUA
1GAIA	2MTAC	1AK0A	1MPGA
1XFFA	2PIAA	1AH7A	1IIBA
1GPCA	2RN2A	13PKA	1FDRA
1HLBA	2SCPA	1AMUA	1NBCA
1IDOA	3CHYA	1MPGA	1CIPA
1KTEA	5P21A	1A48A	2TPSA
1LBAA		1A6QA	1JFRA
1LCLA		2RN2A	1A8LA
1MAIA		2TPSA	2GDMA
1POCA		1AYLA	2PIAA
1POTA		1THTA	
1RCFA		1NO3A	
1RECA		1STFI	
1RSYA		2MBRA	

Table 3.11. Regression accuracy for generic category using common PDB IDs between the 618 data set and the 215, 268, and 1363 data sets for the two regression models. The respective training set and test set used for each of these calculations are also presented in the table.

Models	**Generic**		
	Training = Generic 268 Test = Generic 215	Training = Generic 1363 Test = Generic 215	Training= Generic 268 and Generic 1363 Test = Generic 215
RSA= E20	62.2	61.6	62.01
RSA = E6	68.4	68.5	68.5
RSA = FSHP	70.0	70.1	70.0
RSA = FSHP + AA	71.1	71.3	71.3
RSA = AA	71.4	70.2	71.4
RSA = E20 + AA	73.5	73.3	73.6
RSA = E6 + AA	74.6	73.7	74.4
RSA = E20 + E6 + AA	74.6	73.7	74.4
RSA = E20 + E6	68.8	68.7	68.9
RSA = E20 + FSR + FSHP + AA	74.6	73.7	74.1
RSA = E6 + FSR + FSHP + AA	74.7	73.8	74.5
RSA = E20 + E6 + FSR + FSHP + AA	74.8	73.9	74.5

Table 3.12. Regression accuracy for the protein binding category using common PDB IDs between the 618 data set and the 215, 268, and 1363 data sets for the two regression models. The respective training set and test set used for each of these calculations are also presented in the table.

Models	**Protein Binding**		
	Training = Protein Binding 268 Test = Protein Binding 215	Training = Protein Binding 1363 Test = Protein Binding 215	Training= Protein Binding 268 and Protein Binding 1363 Test = Protein Binding 215
RSA= E20	60.9	60.9	60.9
RSA = E6	60.9	60.9	60.9
RSA = FSHP	75.1	75.0	75.0
RSA = FSHP + AA	74.5	76.2	74.5
RSA = AA	73.4	73.7	73.4
RSA = E20 + AA	74.8	76.0	75.6
RSA = E6 + AA	75.2	77.3	76.7
RSA =E20 + E6 + AA	75.0	77.2	76.3
RSA = E20 + E6	60.9	62.3	60.9
RSA = E20 + FSR + FSHP + AA	76.0	76.2	77.0
RSA = E6+ FSR + FSHP + AA	75.9	76.7	77.9
RSA = E20 + E6 +FSR +FSHP + AA	76.2	76.9	77.3

Table 3.13 is the regression accuracy for oligomer category. Similar to the previous calculations, the oligomer category is the least well-predicted group because accuracies are consistently lower for each of the related models. The highest accuracy for the oligomer category is 73.6% for the RSA = E20 + E6 + FSR + FSHP + AA model using oligomers from 268 as the training and oligomers from 215 as the test set.

Table 3.13. Regression accuracy for the oligomer category using common PDB IDs between the 618 data sets and the 215, 268, and 1363 data sets for the two regression models.

Models	**Oligomer**		
	Training = Oligomer 268 Test = Oligomer 215	Training = Oligomer 1363 Test = Oligomer 215	Training= Oligomer 268 and Oligomer 1363 Test = Oligomer 215
RSA= E20	61.5	60.4	61.2
RSA = E6	68.3	68.5	68.3
RSA = FSHP	67.0	67.1	67.1
RSA = FSHP + AA	69.4	69.4	68.5
RSA = AA	72.4	69.4	68.4
RSA = E20 + AA	73.7	72.1	72.6
RSA = E6 + AA	73.6	73.1	73.5
RSA =E20 + E6 + AA	73.6	73.0	73.4
RSA = E20 + E6	68.9	68.8	68.8
RSA = E20 + FSR + FSHP + AA	73.1	72.1	72.9
RSA = E6+ FSR + FSHP + AA	73.6	73.1	73.5
RSA = E20 + E6 +FSR +FSHP + AA	73.6	73.0	73.4

Table 3.14 represents regression accuracies for the generic category excluding PDB IDs categorized as protein binding and oligomers. The key difference between Table 3.12 generic accuracy and Table 3.14 is that Table 3.12 includes protein binders and oligomers as part of the list. In contrast, for the calculations presented in Table 3.14, these two groups are removed. The highest accuracy for the generic category without the

protein binders and oligomers categories is 76.0% for the RSA = E20 + E6 + FSR + FSHP + AA model.

Table 3.14. Regression accuracy for generic category excluding protein binding and oligomer categories using common PDB IDs between the 618 data sets with 215, 268 and 1363 data sets for the two regression models. The respective training set and test set used for each of these calculations are also presented in the table.

Models	**Generic – Protein Binding - Oligomer**		
	Training = Generic – Protein Binding – Oligomer (268) Test = Generic – Protein Binding – Oligomer (215)	Training = Generic – Protein Binding – Oligomer (1363) Test = Generic – Protein Binding – Oligomer (1363)	Training= Generic – Protein Binding – Oligomer (268 and 1363) Test = Generic – Protein Binding – Oligomer (215)
RSA= E20	64.0	63.5	64.1
RSA = E6	69.9	69.9	69.8
RSA = FSHP	70.7	70.9	70.8
RSA = FSHP + AA	71.8	72.1	72.1
RSA = AA	71.9	70.7	70.7
RSA = E20 + AA	74.6	74.3	74.4
RSA = E6 + AA	75.4	74.7	75.4
RSA =E20 + E6 + AA	75.4	74.7	75.4
RSA = E20 + E6	70.1	70.1	70.0
RSA = E20 + FSR + FSHP + AA	75.6	74.9	75.7
RSA = E6+ FSR + FSHP + AA	75.9	75.4	75.9
RSA = E20 + E6 +FSR +FSHP + AA	76.0	75.4	75.9

4. Discussion

The aggregate and correlation plots of the first of all of the data sets used for this work, the 268 training set, were presented in previous work by the Lustig group (Mishra, 2010). The earlier calculations derived by manually downloaded BLAST files for each PDB ID in the 268 data set has been presented for comparison and validation purposes of the automatically downloaded BLAST output files. The earlier manually presented data and the automated results presented in the results section of this thesis were identical, confirming the validity of the automated system. The automation of downloading BLAST files was then applied toward development of a larger data set. The 1363 training set filled the need for a larger training set. The 1363 set presented trends for all the distribution and correlation plots similar to both the automated and manual 268 training set. Also similar to the 268 training data set, the 1363 aggregate plots displayed the characteristics of two major regions for sequence homology parameters when plotted against inverse Cα packing density.

4.1 Prediction of RSA

The primary focus of this study was to accurately predict solvent accessibility of a given protein residue using a sequence qualitative predictor. The significance of linear regression in conjunction with the amino acid as qualitative predictors lies in the fact that, with a very limited number of sequence homology parameters, one can reasonably predict the likelihood of a residue being either buried or on the surface as a part of binary classification. The prediction accuracy ranged from 73 - 78% with models, and the combination of E20, E6, FSR, FSHP, and AA resulted in the highest accuracy achieved

for most of the subset of calculations performed. The direct introduction of secondary subclass information as qualitative predictors did not improve prediction results. The sequence qualitative predictors directly introduce the actual query information into the analysis. By themselves, they offer significant prediction accuracy. However, our first attempts at nearest neighbor analysis using query sequence information, at least implicitly in averaging flanking of RSA values, did not improve accuracy (Nepal, R. and Lustig, B. San Jose State University, San Jose, CA. Unpublished work, 2011). This approach is a common one in k-nearest neighbor analysis (Joo et al., 2012; Sim et al., 2006).

There appears to be a fundamental limitation of prediction accuracy for solvent-accessible residues. This involves having to deal with the association of solvent accessibility with quaternary structure. Similar intrinsic limitation is also noticed in secondary structure prediction (Kihara, 2005; Rost, 2001) with respect to tertiary structure. Though we calculated a 40% alignment score as a threshold, this is comparable to a sequence identity score of 40% (Yeh, 2005). But truncating sequences has its own problems including losing valuable information about the nature of certain substituted residues. Although sequence alignment helps identify differences between protein pairs of similar and non-similar structures in high sequence identity (>40% for long alignment), the signal gets less clear in 20–35% sequence identity (Jaroszewski et al., 2002; Rost, 1999; Schwarz et al., 2010). This blurring should have an impact on solvent accessibility.

There have been some additional improvements in solvent accessibility predictions using support vector machines (Adamczak et al., 2004) and other learning-

based approaches, such as the random forest method to determine specific accessible surface area (Pugalenthi et al., 2012). These calculations still remain very computationally intensive and somewhat obscure in the physical interpretation of individual parameters. However, even under optimal threshold RSA criteria, the overall binary prediction limits remain just at or below the 78% accuracy. One parameter that could have an impact is the quality of a learning set. Assuming this source of error is addressed, the intrinsic limitations from coupling of local secondary and higher orders of 3D structure will likely still remain.

The most significant limitation for the prediction of residue solvent accessibility may be attributed to the coupling between residue surface accessibility and protein-protein contacts including quaternary structure. Although the majority of hydrophobic residues are found buried in the core of the protein structure, there are some hydrophobic residues on the surface of the protein involved in interaction with other proteins (Yan et al., 2008). Earlier work by the Lustig group presented a linear correlation between query hydrophobicity and inverse packing density in most of the Major Region I (Liao et al., 2005). On the other hand, 10% of Major Region II query residues were identified as strongly hydrophobic.

Although the accuracies calculated by linear regression model as described in this thesis are slightly lower, it is comparable to the most utilized RSA accuracy calculation reported in the literature (Adamczak et al., 2004). The former uses a complicated system built upon neural networks that involve multilayered feed. A continuous approximation of the real-value RSA using nonlinear regression was used with several feed-forward and

recurrent neural networks that were then combined into a consensus predictor. We employed a simple two-step linear regression method for RSA prediction. Also, it was reported that the use of Position-Specific Iterative BLAST (PSI-BLAST) resulted in better accuracy. Our calculations resulted in similar accuracies with PSI-BLAST and did not improve the accuracy predictions.

We have found that a very limited number of parameters can result in significant prediction accuracy. These parameters include direct descriptor of actual sequence and the various Shannon entropies associated with their substitution. The inclusion of 20-point sequence entropy displayed the flexibility of a given amino acid to change or mutate (Koehl and Levitt, 2001). Also noteworthy is that the addition of parameters involving the classification of strongly hydrophobic or small residues add some incremental value to the prediction. However, once one achieves accuracy approaching mid 70%, additional parameter components add very little incrementally to the prediction accuracy.

Various surface accessibility prediction methods applying the two-state (buried and on the surface) have been recently developed. The use of SVM has been explored to improve solvent accessibility prediction accuracies (Kim and Park, 2003; Wang et al., 2007; Yuan et al., 2002). Typically, SVM constructs extended representation of data and then classifies them into groups. This requires the use of a training set to inform the boundaries of the classifier. Another method extensively applied to solvent accessibility prediction is the utilization of neural networks (Adamczak et al., 2004; Ahmad and Gromiha, 2002; Kim and Park, 2003; Rost and Sander, 1994). In neural networks

methods, RSA is predicted using a non-linear regression method instead of a classification method. Unlike in SVM methods, here a continuous approximation and evaluation of the real-value RSA is produced instead, imposing an arbitrary threshold to the RSA (Adamczak et al., 2004). One such application in surface accessibility prediction is the fuzzy k-nearest method applied to sequence information (FKNN) by Sim et al. (2006); Joo et al. (2012) presented an additional application of the nearest neighbor method where a database is constructed based on sequence information of residues and its neighbors. Accuracy in all of these methods typically stays near 80%, and this is consistent with the intrinsic problem at hand.

From a limited set of known quaternary contacts (Do, S.and Lustig, B. San Jose State University, San Jose, CA. Unpublished work, 2010) it was concluded that RSA prediction of residues on the surface is problematic. The RSA prediction of residues is challenging primarily due to the presence of hydrophobic patches on the surface of the protein. However, it is known in general that 50% of the globular protein's surface is non-polar, making "hydrophobic patches" inevitable even if protein does not interact with other proteins (Eisenhaber and Argos, 1996; Lins et al., 2003). Although the majority of hydrophobic residues are found buried deep within a protein structure, some are present on the surface of the protein interacting with other complexes. To address this issue in solvent accessibility prediction, Bahadur et al. (2004) described a method in which surface residues are divided into two groups: the "core" and the "rim" set. This core is not necessarily what is descriptive of the core in buried residues of the folded protein. The core residues present in surface-accessible patches include interfaces with quaternary

interaction, and a mere presence of one buried interface atom is enough to categorize a given residue as a core residue (Chakrabarti and Janin, 2002).

The incorporation of 618 categorical PDB IDs (Petit et al., 2007) was performed in an attempt to improve RSA prediction accuracies. These proteins are in context of an algorithm to predict, from X-ray structure, functionally relevant patches on the surface of various classes of proteins. It was observed that, out of all the calculations performed, our method consistently resulted in a better prediction for the protein binding category better than any other group, whereas the oligomer group dependently resulted in lower accuracies. The difference between the highest protein binding category and the highest oligomer category was about 4.5%. Noteworthy, the remaining of the 618 PDB IDs excluding protein binding PDB IDs, and oligomeric IDs, resulted in prediction accuracies better than the oligomeric category, but not as good as protein binding. Here the difference between protein binding and this group was around 2%.

In this thesis we have outlined a reliable RSA prediction method, and further enhancements in this scheme have a potential to be a novel simple binary RSA predictions method as a gold standard for the field. Incorporation of a larger training data set (i.e., 1363) did not yield a drastic improvement over RSA prediction. Unlike Joo et al. (2012), who claimed the larger data set improved results drastically, our results do not indicate such. The original size of the data set for the 1363 PDB IDs in the 1363 training set was 6511 proteins (Bondugula et al., 2011), and similar PISCES culling parameters were applied to this list as stated in the Joo et al. paper. After the application of PISCES, the original list of 5157 (subset of the 6511 PDBs that were part of NCBI PDB library)

was reduced to 1363 of structurally diverse set. Even though, Joo et al. (2012) found 5717 proteins, after application of the culling service PISCES, it is hard to imagine that the list is not structurally redundant.

Our results have indicated that dividing the proteins into various structural categories holds itself as a promising new direction for RSA predictions. The role of solvent accessibility in protein binding categories and oligomers needs to be further investigated to discover why the latter results in poor prediction accuracy while the former does not. Currently for this work, the incorporation of structural information (i.e., protein binding, oligomers) to the existing data sets (268, 1363, and 215) could only be applied to PDB IDs that were present in the 618 data set. As a future direction to the project, a reliable method to apply structure information to every PDB ID in the existing data sets needs to be assimilated. Once this has been done, the RSA prediction calculations need to be recalculated, and any changes observed could be noteworthy. Also, multiple stages have displayed some level of advantage in terms of RSA prediction with SVM and linear regressions (Adamczak et al., 2005). A second-stage implementation of the existing first-stage qualitative predictor methodology may present itself as a new improvement to the existing systems.

5. Future Studies

Additional studies to further the work described in this thesis are listed below:

- Investigate the reasons why the protein binding category consistently results in higher prediction accuracy, whereas the oligomer category yields a lower range of prediction accuracy.
- Protein binding and oligomeric classifications should be made for all of the three different data sets (268, 1363, and 215).
- A second stage regression method should be incorporated into the existing regression approach. Our preliminary results indicated the incorporation of a distant homolog as a second filter yielded higher accuracy; this approach needs to be further analyzed.
- The possibility of application of logistic regression where an outcome is predicted on the basis of categorical dependent variable in the presence of one or more predictor variables should be explored for better accuracy predictions.
- Further in-depth exploration of the current method should be implemented to recognize amino acid residues and RSA values that are most mispredicted with the existing method.
- Further investigate a larger training NACCESS threshold than the 20 that is currently being used.

6. Conclusions

Query-based qualitative predictors with protein sequence information were utilized to predict protein residue solvent accessibility. An automated system to download, integrate, and analyze various homology-based parameters and calculations was developed. The manual and automated characterization were deemed to be identical, thereby validating the later method. A novel and larger training data set (1363 training) was developed. The distribution plots of both the 268 and 1363 training data sets displayed a bimodal frequency distribution of residues, indicating the presence of highly hydrophobic residues on the surface. This is consistent with the notion of intrinsic limitations in predicting surface-accessible residues with only one chain.

A total of 12 main regression models utilizing various combinations of the two types of entropies (E20 and E6), homology-based parameters (FSR, FSHP) as quantitative predictors, and direct 20 amino acid information as qualitative predictors were created. Our results indicate consistently that E20 + E6 + FSR + FSHP + AA resulted in the highest accuracy compared to all the other models for all the tests performed. Interestingly, 6-point entropy (E6) with the qualitative predictor (AA) resulted in better prediction than the use of 20-point entropy (E20). In fact, in some sets of linear regression accuracy calculations, E6 + AA did almost as well as the best model. And breaking the FSR qualitative predictor into its two individual components (FA and FG) did not improve prediction accuracy. The incorporation of a larger training data set did not improve the accuracy prediction of the test set but actually resulted in accuracies comparable to the smaller training data set.

Categorical regressions, where the proteins are divided into groups such as protein binding and oligomer, bring about some interesting information. The linear regression method imposed in this thesis seemed to consistently yield higher prediction accuracy for the protein binding category. On the other hand, the oligomeric category resulted in slightly lower prediction accuracies.

Bibliography

Adamczak, R.; Porollo, A.; Meller, J. *Proteins* **2004**, 56, 753-767.

Adamczak, R.; Porollo, A.; Meller, J. *Proteins* **2005**, 59, 467-475.

Ahmad S.; Gromiha, M.M. *Bioinformatics* **2002**, 18, 819-824

Bahadur, R. P.; Chakrabarti, P.; Rodier, F.; Janin J. *J. Mol. Biol.* **2004**, 336, 943-955.

Berezovsky. I. N.; Trifonov, E. N. *J. Mol. Biol.* **2001**, 307, 1419-1426.

Berman, H. M.; Westbrook, J.; Feng, Z.; Gilliland, G.; Bhat, T. N.; Weissig, H.; Shindyalov, I. N.; Bourne, P. E. *Nucl. Acids. Res.* **2000**, 28, 235-242.

Berman, H. *Structure*. **2008**, 16, 16-18.

Bondugula, R.; Wallqvist, A.; Lee, M. S. *Protein Eng. Des. Sel*. **2011**, 24, 455-461.

Carugo, O. *Protein Eng.* **2000**, 13, 607-609.

Chakrabarti, P.; Janin, J. *Proteins* **2002**, 47, 334-343.

Chayen, N. E.; Saridakis, E. *Nature Methods*. **2008**, 5, 147-153.

Daggett, V.; Fersht, A. *Nature Rev. Mol. Cell Biol.* **2003**, 4, 497-502.

Dale, G. E.; Oefner, C.; d'Arcy, A. *J. Struct. Biol.* **2003**, 142, 88-97.

Eisenhaber, F.; Argos, A. *Protein Eng.* **1996**, 9, 1121-1133.

Eswar, N.; Marti-Renom, M. A.; Webb, B.; Madhusudhan, M. S.; Eramian, D.; Shen, M.; Pieper, U.; Sali. *Curr. Protoc. Bioinformatics* **2006**, Supplement 15, 5.6.1-5.6.30.

Gluehmann, M.; Zarivach, R.; Bashan, A.; Harms, J.; Schluenzen, F.; Bartels, H.; Agmon, I.; Rosenblum, G.; Pioletti, M.; Auerbach, T.; Avila, H.; Hansen, H. A. S.; Franceschi, F.; Yohnath, A. *Methods* **2001**, 25, 292-302.

Graebsch, A.; Roche, S.; Kostrewa, D.; Soding, J.; Niessing, D. *PLos One*. **2010**, 5 10.

Guharoy, M.; Chakrabarti, P. *Natl. Acad. Sci. USA* **2005**, 102, 15447-15452.

Hellevik, O. *Qual. Quant.* **2009**, 43, 59-74.

Hsieh, M.; Collins, E. D.; Blomquist, T.; Lustig, B. *J. Biomol. Struct. Dyn.* **2002**, 20, 243-251.

Hubbard, S. J.; Campbell, S. F.; Thornton, J. M. *J. Mol. Biol.***1991**, 220, 507-530.

Hubbard, S. J.; Thornton, J. M. *NACCESS, Computer Program, Department of Biochemistry and Molecular Biology, University College London.*http://www.bioinf.manchester.ac.uk/naccess/, 1993.

Jaroszewski, L.; Li, W.; Godzik, A. *Protein Sci*. **2002**, 11, 1702-1713.

Joo, K.; Lee, S. J.; Lee, J. *Proteins* **2012,** 80, 1791-1797.

Kallblad, P.; Dean, P. M. *Proteins* **2004**, 56, 693-703.

Kihara, D. *Protein Sci.* **2005**, 14, 1955-1963.

Kim, H.; Park, H. **2003**, *Protein Eng*. **2003**, 16, 553-560.

Koehl, P.; Levitt, M. *Proc. Natl. Acad. Sci. USA* **2002**, 99, 1280-1285.

Kutner, M. H, Nachshelm, C. J, Neter, J. *Applied Linear Statistical Models*; McGraw-Hill, New York, CA, 2004, Vol. 5, Chapter 8.

Liao, H.; Yeh, W.; Chiang, D.; Jernigan, R. L.; Lustig, B. *Protein Eng.* **2005**, *18,* 59–64.

Lins, L.; Thomas, A.; Brasseur, R. *Protein Sci.***2003**, 12, 1406-1417.

Mishra, R. *Characterization of Protein Residue Structural Accessibility Using Sequence Entropy*; M.S. Thesis, San Jose State University, San Jose, CA, 2010.

Mizianty, J.; Kurgan, L. *Bioinformatics* **2011**, 27, i24-i33.

Mizianty, M. J.; Kurgan, L. A. *Protein Pept. Lett.* **2012**, 19, 40-49.

Naderi-Manesh, H.; Sadeghi, M.; Arab, S.; Movahedi, A. A. M. *Proteins* **2001**, 42, 452-459.

National Center for Biotechnology Information (NCBI), Protein Blast (BLASTP), http://www.ncbi.nlm.nih.gov. Accessed 2011.

Oliveira, L.; Paiva, A. C. M.; Vriend, G. *ChemBioChem.* **2002**, 3, 1010-1017.

Panchenko, A. R.; Kondrashov, F.; Bryant, S. *Prot. Sci.* **2004**, 13, 884-892.

Petrova, N. V.; Wu, C. H. *BMC Informatics* **2006**, 7, 313-323.

Pettit, F. K.; Tsa, A.; Bowie, J. U. *J. Mol. Biol.* **2007**, 369, 863- 879.

Porollo, A.; Meller, J. *Proteins* **2007**, 66, 630-645.

Poupon, A.; Mornon, J. P. *FEBS Lett*. **1999**, 452, 283-289.

Pugalenthi, G.; Kandaswamy, K. K.; Chou, K.; Vivekanandan, S.; Kolatkar, P. *Protein Pept. Lett*. **2012**, 19, 50-56.

Richardson, C. J.; Barlow, D. J. *Prot. Engr*. **1999**, 12, 1051-1054.

Rose, D. A.; Nepal, R.; Mishra, R.; Lau, R.; Gholizadeh, S.; Lustig, B. *Conference in Twenty-Secondary International Workshop on Database and Expert Systems Applications, DEXA*, Toulouse, France, September 29, 2011; Morvan, F.; Tjoa, A. M.; Wagner, R. R., Eds.; IEEE Computer Society, Los Alamitos, 2011, 70-74.

Rost, B. J. *Struct. Biol.* **2001**, 134, 204-218.

Rost, B.; Sander, C. *Proteins* **1994**, 20, 216-226.

Savchenko, A.; Yeh, A.; Khachatryan, A.; Evdokimova, E.; Pavlova, M.; Semes, A.; Northey, J.; Beasley, S.; Lan, N.; Das, R.; Gerstein, M.; Arromith, C.H.; Edwards, A.M. *Proteins* **2003**, 50, 392-399.

Schwarz, R.F.; Fletcher, Wi.; Forster, F.; Merget, B.; Wolf, M.; Schultz, J.; Markowetz, F. *PLoS One.* **2010**, 5, 12.

Shenkin, P. S.; Erman, B.; Mastrandrea, L.D. **1991**, *Proteins*, 11, 297-313.

Sim, J.; Kim, S.Y.; Lee, J. *Bioinfomatics*. **2006**, 21, 2844-2849.

Structural Classification of Proteins (SCOP), http://scop.mrc-lmb.cam.ac.uk/scop/. Accessed 2011.

Valdar, W. S. J. *Proteins* **2002**, 48, 227-241.

Wagner, M.; Adamczak, R.; Porollo, A.; Meller, *J. J. Comput. Biol*. **2005**, 12, 355-369.

Wang, G.; Roland L. Dunbrack, *J. Bioinformatics* **2003**, 19, 1589–1591.

Wang, J.Y.; Lee, H.; Ahmad, S. *Proteins* **2007**, 68, 82-91.

Yan, C.; Terribilini, M.; Wu, F.; Jernigan, R. L.; Dobbs, D.; Honavar, V. *BMC Bioinformatics* **2006**, 7, 262-271.

Yan, C.; Wu, F.; Jerrigan, R. L.; Dobbs, D.; Honavar, V. *Prot J*. **2008**, 27, 59-70.

Yeh, W. *Detailed analysis of protein sequence entropy, hydrophobicity, and flexibility;* MS Thesis, San Jose State University, San Jose, 2005.

Yuan, Z.; Burrage, K.; Mattick, J.S. *Proteins* **2002**, 48, 566-570.

Appendices

A. Program Listings

```
################################################################################
# Author: Reecha Nepal
# Date: May 6, 2012
# Purpose: This script downloads the blast results for all pdb names in a
# given file. To run this script do the following:
#    python download_blast.py pdb_names.txt
#    or
#    python download_blast.py 119LA
# The output files will be created in a directory called "blast".
# File: download_blast.py
################################################################################

import os
import sys

################################################################################
# Returns the blast results for the given GI number.
################################################################################
def download_blast_for_gi_number(gi_number):
    from Bio.Blast import NCBIWWW
    blast_result_handle = NCBIWWW.qblast('blastp', 'nr', gi_number, \
                                         format_type="Text", alignments="1000",
                                         descriptions="10000",
                                         hitlist_size="10000")
    blast_text = blast_result_handle.read()
    blast_result_handle.close()
    return blast_text

################################################################################
# Reads the fasta file for the given pdb name.
################################################################################
def read_fasta_file_for_pdb_name(pdb_name):
    file_name = os.path.join("fasta", pdb_name + ".fasta")
    if not os.path.exists(file_name):
        print "Error: Couldn't find file:", file_name
        print "Did you run the download_fasta.py script?"
        sys.exit(-1)

    input_file_handle = open(file_name, "r")
    input_data = input_file_handle.read()
    input_file_handle.close()
    return input_data

################################################################################
# Gets the output path for the given pdb_name. For example, if the PDB name
# is 1HGXA then the output path would be:
#     blast/1HGXA.txt
################################################################################
def get_output_path(pdb_name):
    # put the file in a directory named "blast"
    if not os.path.exists("blast"):
        os.makedirs("blast")

    output_path = os.path.join("blast", pdb_name + ".txt")
    return output_path

################################################################################
# Saves the blast data with the fasta data at the top.
################################################################################
```

```
def save_blast_data(fasta_data, blast_data, output_path):
    out_file_handle = open(output_path, "wb")
    out_file_handle.write(fasta_data)
    out_file_handle.write(blast_data)
    out_file_handle.close()

################################################################################
# Gets the GI number from a fasta file.
################################################################################
def get_gi_number_from_fasta(fasta_data):
    # The fasta_data should look like this:
    #    >gi|1827785|pdb|1HGX|A Chain
    words = fasta_data.split("|")

    # If there are less than 4 words then this fasta file has errors in it.
    if len(words) < 4:
        print "Error: The fasta file for this PDB has errors in it."
        sys.exit(-1)

    return words[1]

################################################################################
# Get a list of pdb names from the a file.
################################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython download_blast.py pdb_names.txt\n" \
              "\tor python download_blast.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])

    return pdb_list

################################################################################
# Runs the main script.
################################################################################
def run(pdb_name):
    output_path = get_output_path(pdb_name)
    if os.path.exists(output_path):
        return

    fasta_data = read_fasta_file_for_pdb_name(pdb_name)
    gi_number = get_gi_number_from_fasta(fasta_data)
    blast_data = download_blast_for_gi_number(gi_number)
    save_blast_data(fasta_data, blast_data, output_path)

################################################################################
# The main function, this gets run first when the program is run from the
# command line.
################################################################################
if __name__ == "__main__":
    pdb_list = get_pdb_list()
    for index in range(0, len(pdb_list)):
        pdb_name = pdb_list[index]

        # Show how much is done
```

```
        percent_done = (index + 1.0) / len(pdb_list)
        percent_done = int(percent_done * 100.0)
        print index + 1, percent_done, "%", pdb_name

        run(pdb_name)

###############################################################################
# Author: Reecha Nepal
# Date: May 6, 2012
# Purpose: This script downloads the fasta files for all pdb names in a given
# file. To run this script do the following:
#    python download_fasta.py pdb_names.txt
#    or
#    python download_fasta.py 119LA
# The output files will be created in a directory called "fasta".
# File: download_fasta.py
###############################################################################

from urllib import urlopen
import os
import time
import sys

###############################################################################
# Download and return the fasta file for the given GI number.
###############################################################################
def download_fasta_for_gi_number(gi_number):
    url = "http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?" \
          "db=nucleotideandid=" + gi_number + "andrettype=fasta"
    url_file_handle = urlopen(url)
    fasta_data = url_file_handle.read()
    url_file_handle.close()
    return fasta_data

###############################################################################
# Gets the output path for the given pdb_name. For example, if the PDB name
# is 1HGXA then the output path would be:
#     fasta/1HGXA.fasta
###############################################################################
def get_output_path(pdb_name):
    # put the file in a directory named "blast"
    if not os.path.exists("fasta"):
        os.makedirs("fasta")

    output_path = os.path.join("fasta", pdb_name + ".fasta")
    return output_path

###############################################################################
# Save the fasta file in a directory called fasta.
###############################################################################
def save_fasta_data(fasta_data, output_path):
    out_file_handle = open(output_path, "wb")
    out_file_handle.write(fasta_data)
    out_file_handle.close()

###############################################################################
# Checks if the fasta data is valid. A valid fasta data should look like this:
#    >gi|1827785|pdb|1HGX|A Chain
# If there's a server error then we sometimes get data that looks like this:
#    Error:Cannot connect to database
###############################################################################
def fasta_data_is_valid(fasta_data):
    fasta_data_string = fasta_data.decode("utf-8")
    words = fasta_data_string.split("|")
    if len(words) < 4:
        return False
    else:
```

```
        return True

################################################################################
# Get a list of pdb names from a file.
################################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython download_fasta.py pdb_names.txt\n" \
              "\tor python download_fasta.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])

    return pdb_list

################################################################################
# Gets the GI number for the given PDB name.
################################################################################
def get_gi_number(pdb_name):
    file_name = os.path.join("gi_number", pdb_name + ".txt")
    if not os.path.exists(file_name):
        print "Error: Couldn't find file:", file_name
        print "Did you run the download_gi_number.py script?"
        sys.exit(-1)

    input_file_handle = open(file_name, "r")
    input_data = input_file_handle.read()
    input_file_handle.close()
    return input_data.strip()

################################################################################
# Runs the main script.
################################################################################
def run(pdb_name):
    output_path = get_output_path(pdb_name)
    if os.path.exists(output_path):
        return

    gi_number = get_gi_number(pdb_name)

    retry_count = 0
    while True:
        # Download and save the fasta file
        fasta_data = download_fasta_for_gi_number(gi_number)
        if fasta_data_is_valid(fasta_data):
            if retry_count > 0:
                print "Downloading fasta data succeeded after", \
                        retry_count, "tries"
            save_fasta_data(fasta_data, output_path)
            break

        # If there was an error then retry up to 8 times.
        if retry_count == 0:
            print "Warning: Downloading fasta data failed, retrying"
        elif retry_count > 8:
            print "Error: Unable to download fasta file for pdb:", \
                    pdb_name, "quitting."
```

```
            sys.exit(-1)
        else:
            print "Retrying", retry_count
        retry_count = retry_count + 1

        # Wait 0.1 seconds before trying again incase we're overloading
        # the server.
        time.sleep(0.1)

################################################################################
# The main function, this gets run first when the program is run from the
# command line.
################################################################################
if __name__ == "__main__":
    pdb_list = get_pdb_list()

    for index in range(0, len(pdb_list)):
        pdb_name = pdb_list[index]

        # Show how much is done
        percent_done = (index + 1.0) / len(pdb_list)
        percent_done = int(percent_done * 100.0)
        print index + 1, percent_done, "%", pdb_name

        run(pdb_name)

################################################################################
# Author: Reecha Nepal
# Date: July 24, 2012
# Purpose: This script downloads the GI number for all pdb names in a given
# file. To run this script do the following:
#    python download_gi_number.py pdb_names.txt
#    or
#    python download_gi_number.py 119LA
# The output files will be created in a directory called "gi_number".
# File: download_gi_number.py
################################################################################

from urllib import urlopen
from xml.dom.minidom import parseString
import os
import time
import sys

################################################################################
# Download and return the GI number file for the given pdb name.
################################################################################
def download_gi_number_for_pdb_name(pdb_name):
    url = "http://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?" \
          "db=proteinandterm=" + pdb_name + "andretmode=xml"
    url_file_handle = urlopen(url)
    gi_number_data = url_file_handle.read()
    url_file_handle.close()
    return gi_number_data

################################################################################
# Gets the output path for the given pdb_name. For example, if the PDB name
# is 1HGXA then the output path would be:
#     gi_number/1HGXA.txt
################################################################################
def get_output_path(pdb_name):
    # put the file in a directory named "blast"
    if not os.path.exists("gi_number"):
        os.makedirs("gi_number")

    output_path = os.path.join("gi_number", pdb_name + ".txt")
    return output_path
```

```
################################################################################
# Save the GI number in a directory called gi_number.
################################################################################
def save_gi_number_data(gi_number, output_path):
    out_file_handle = open(output_path, "wb")
    out_file_handle.write(gi_number)
    out_file_handle.close()

################################################################################
# Checks if the GI number data is valid. Valid data should look like this:
#    <eSearchResult>
#    <Count>1</Count>
#    <RetMax>1</RetMax>
#    <RetStart>0</RetStart>
#    <IdList>
#    <Id>157829547</Id>
#    </IdList>
#    <TranslationSet/>
#    <QueryTranslation/>
#    </eSearchResult>
# If there's a server error then we sometimes get data that looks like this:
#    Error:Cannot connect to database
# If the data is valid then returns the GI number.
################################################################################
def parse_gi_number_from_data(data):
    string = data.decode("utf-8")
    if string.find("eSearchResult") == -1:
        return -1

    dom = parseString(data)
    id_list = dom.getElementsByTagName("Id")
    if not id_list or len(id_list) == 0:
        return -1
    return id_list[0].childNodes[0].nodeValue

################################################################################
# Get a list of pdb names from a file.
################################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython download_gi_number.py pdb_names.txt\n" \
              "\tor python download_gi_number.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])

    return pdb_list

################################################################################
# Runs the main script.
################################################################################
def run(pdb_name):
    output_path = get_output_path(pdb_name)
    if os.path.exists(output_path):
        return
```

```
    retry_count = 0
    while True:
        # Download and save the gi number
        gi_number_data = download_gi_number_for_pdb_name(pdb_name)
        gi_number = parse_gi_number_from_data(gi_number_data)
        if gi_number != -1:
            if retry_count > 0:
                print "Downloading gi number data succeeded after", \
                        retry_count, "tries"
            save_gi_number_data(gi_number, output_path)
            break

        # If there was an error then retry up to 8 times.
        if retry_count == 0:
            print "Warning: Downloading GI number failed, retrying"
        elif retry_count > 8:
            print "Error: Unable to download GI number for pdb:", \
                    pdb_name, "quitting."
            sys.exit(-1)
        else:
            print "Retrying", retry_count
        retry_count = retry_count + 1

        # Wait 0.5 seconds before trying again incase we're overloading
        # the server.
        time.sleep(0.5)

################################################################################
# The main function, this gets run first when the program is run from the
# command line.
################################################################################
if __name__ == "__main__":
    pdb_list = get_pdb_list()

    for index in range(0, len(pdb_list)):
        pdb_name = pdb_list[index]

        # Show how much is done
        percent_done = (index + 1.0) / len(pdb_list)
        percent_done = int(percent_done * 100.0)
        print index + 1, percent_done, "%", pdb_name

        run(pdb_name)
```

```
################################################################################
# Author: Reecha Nepal
# Date: July 18, 2012
# Purpose: Convert SCOP (Structural Classification of Proteins) ids to PDB ids.
# File: SCOPid_to_PDBid.py
################################################################################

input_file = open("260 Similarity log.txt", "r");
pdb_names = []
pdb_rejected_mapping = { }
for line in input_lines:
  # line is something like "reject 1CSEE 2PRKA  38"
  words = line.split(" ")
  # words is something like ["reject", "1CSEE", "2PRKA", "38"]
  number_of_words = len(words)
  if number_of_words >= 4:
    current_pdb_name = words[1]
    rejected_pdb_name = words[2]
    # at this point, current_pdb_name is something like "1CSEE"
    if not current_pdb_name in pdb_names:
      # this is a new pdb name
      pdb_names.append(words[1])
```

```
      pdb_rejected_mapping[current_pdb_name]= [rejected_pdb_name]
    else:
      # give me the list of rejected pdb names for current_pdb_name
      rected_pdb_names = pdb_rejected_mapping[current_pdb_name]
      # add the rejected pdb name
      rected_pdb_names.append(rejected_pdb_name)
      # update our mapping
      pdb_rejected_mapping[current_pdb_name] = rected_pdb_names

for pdb_name in pdb_names:
  print(pdb_name)
  if False:
    print(" The rejected pdbnames are ")
    # print the rejected pdb names for this pdb_name
    list_of_rejected_names = pdb_rejected_mapping[pdb_name]
    for a in list_of_rejected_names:
      print(" ")
      print(a)

  print("\n")

#############################################################################
# Author: Reecha Nepal
# Date: July 18, 2012
# Purpose: Takes pdb name from the 6511 total Bondugula set, checks if each of
# those proteins are listed in pdb website or not, and finally print out an
# output file consisting of just the pdb names found in pdb website
# File: common pubs in the Bondugula set vs pdb library.py
#############################################################################

# step 1A

# list of pdb names from the Bondugulla paper
input_file_Bondugulla = open("Bondugulla_pdbid_4letters.txt", "r");
source_pdb_lines = input_file_Bondugulla.readlines()
# There's only one line in the file, so just split that one line
# into words and save it into the variable source_pdb_list
source_pdb_first_line = source_pdb_lines[0]
source_pdb_list= source_pdb_first_line.split(", ")



# step 1B
# list of all the pdb names from the RCSB database
input_file_pdb = open("list of all pdbs in RCSB as of July8,2011.txt", "r");
master_pdb_list = input_file_pdb.readlines()
# There's only one line in the file, so just split that one line
# into words and save it into the variable source_pdb_list
master_pdb_first_line = master_pdb_list[0]
master_pdb_list = master_pdb_first_line.split(", ")


#Step 2
matched_pdb_name_list= []
for current_pdb in source_pdb_list:
    if current_pdb in master_pdb_list:
        matched_pdb_name_list.append(current_pdb)


#Step 3
output_file = open("Common pdbs between the Bondugulla set and pdb library.txt", "w")
for matched_pdb_name in matched_pdb_name_list:
    output_file.write(matched_pdb_name )
    output_file.write("\n ")

output_file.close()
```

```
###############################################################################
# Author: Reecha Nepal
# Date: July 22, 2012
# Purpose: This script calculates packing density.
# To run this script do the following:
#    python calculate_density.py pdb_names.txt
# The output files will be created in a directory called "density".
#
# This script is adapted from:
#     cif2den.pl written by Radhika Pallavi Mishra
#     pdb2denMOD2-2 written by William Yeh
# File: calculate_density.py
###############################################################################

import os
import sys
import math
import exceptions

# User Specified Variables to Control Analysis and Output
# For each Value, calc's #dist <= Value
TAB_VALUES = [6, 7, 8, 9, 10, 11, 12]

# Must be increasing in value.
# Defines which TAB_VALUES should be printed.
TAB_PRINT = [0, 1, 2, 3, 4, 5, 6]

# Initialize Amino Acid 3-letter to 1-letter associative list
AA_DICTIONARY = {
    'GLY': 'G', 'ALA': 'A', 'VAL': 'V', 'LEU': 'L',
    'ILE': 'I', 'MET': 'M', 'PRO': 'P', 'PHE': 'F',
    'TRP': 'W', 'SER': 'S', 'THR': 'T', 'ASN': 'N',
    'GLN': 'Q', 'TYR': 'Y', 'CYS': 'C', 'LYS': 'K',
    'ARG': 'R', 'HIS': 'H', 'ASP': 'D', 'GLU': 'E'
}

def get_int(str_value):
    try:
        return int(str_value)
    except exceptions.ValueError:
        return 0

def get_float(str_value):
    try:
        return float(str_value)
    except exceptions.ValueError:
        return 0.0


class AminoAcidSeqRes:
    def __init__(self):
        # Should be something like "A"
        self.amino_acid_short = ""
        # FASTA position.
        self.fasta_pos = 0
        # PDB position
        self.pdb_pos = 0

class AlphaC:
    def __init__(self):
        # Amino acid, should be something like "LYS"
        self.amino_acid = ""
        # Short form, should be something like "A"
        self.amino_acid_short = ""
        # (x, y, z) coordinates from ATOM statement
        self.x = 0
        self.y = 0
```

```
        self.z = 0
        # PDB position
        self.pdb_pos = 0

################################################################################
# Extract Amino Acid seq from SEQRES statements.
################################################################################
def extract_amino_acid_seqres(input_lines, i, chain_name):
    array = []

    i = i + 1
    line = input_lines[i]
    while i < len(input_lines) and line != "loop_":
        words = line.split()
        if len(words) > 9 and words[9] == chain_name:
            aa = AminoAcidSeqRes()
            if words[3] in AA_DICTIONARY:
                aa.amino_acid_short = AA_DICTIONARY[words[3]]
            else:
                aa.amino_acid_short = "NA"
            aa.fasta_pos = get_int(words[4])
            if words[6].isdigit():
                aa.pdb_pos = get_int(words[6])
            else:
                aa.pdb_pos = -1
            array.append(aa)
        i = i + 1
        line = input_lines[i]
    return i, array

################################################################################
# Extract alpha-C (x,y,z) from ATOM statements
################################################################################
def extract_alpha_c(input_lines, i, chain_name):
    array = []

    i = i + 1
    line = input_lines[i]
    count_1 = 0
    count_2 = 0
    count_3 = 0
    while i < len(input_lines) and not line.startswith("#"):
        count_1 = count_1 + 1
        words = line.split()
        # Find alpha-Carbon ATOM lines
        if words[0] == "ATOM" and words[3] == "CA":
            count_2 = count_2 + 1
            aa_ref = words[5]
            aa_refi = words[23]
            if aa_ref in AA_DICTIONARY and aa_refi == chain_name:
                count_3 = count_3 + 1
                alphac = AlphaC()
                alphac.amino_acid = aa_ref
                alphac.amino_acid_short = aa_refi
                alphac.x = get_float(words[10])
                alphac.y = get_float(words[11])
                alphac.z = get_float(words[12])
                alphac.pdb_pos = get_int(words[21])
                array.append(alphac)
            # TODO: This line is probably a bug and should be removed.
            i = i + 1
        i = i + 1
        line = input_lines[i]

    return i, array

################################################################################
```

```
# Extracts all the data from the given file.
################################################################################
def extract_all(input_lines, chain_name):
    aa_seqres_array = []
    alphac_array = []
    pdb_name = ""

    i = 0
    while i < len(input_lines):
        line = input_lines[i]
        if line.startswith("_pdbx_poly_seq_scheme.pdb_ins_code"):
            i, array = extract_amino_acid_seqres(input_lines, i, chain_name)
            aa_seqres_array.extend(array)

        line = input_lines[i]
        if line.startswith("_atom_site.pdbx_PDB_model_num"):
            i, array = extract_alpha_c(input_lines, i, chain_name)
            alphac_array.extend(array)

        # extract the PDB name from the header line
        line = input_lines[i]
        if line.startswith("data_"):
            pdb_name = line.split("_")[1]
            pdb_name = pdb_name.lower()
        i = i + 1

    return aa_seqres_array, alphac_array, pdb_name

################################################################################
# Calculate distances and tabulate.
################################################################################
def calculate_densities(alphac_array):
    pos_den_hash = {}

    for i in range(0, len(alphac_array)):
        # Calculate distances
        distances = []
        for j in range(0, len(alphac_array)):
            x = alphac_array[j].x - alphac_array[i].x
            y = alphac_array[j].y - alphac_array[i].y
            z = alphac_array[j].z - alphac_array[i].z
            value = math.sqrt(x * x + y * y + z * z)
            distances.append(math.sqrt(x * x + y * y + z * z))

        # Sort and tabulate according to distance
        tab_count = []
        for tab_value in TAB_VALUES:
            count = 0
            for distance in distances:
                if distance <= tab_value:
                    count = count + 1
            tab_count.append(count)

        # store density values in a hash corresponding to their PDB position
        pos_den_hash[alphac_array[i].pdb_pos] = tab_count

    return pos_den_hash

################################################################################
# Prints the density values to the given file.
################################################################################
def print_densities(cif_pdb_name, aa_seqres_array, pos_den_hash):
    lines = []
    for aa_seqres in aa_seqres_array:
        value = ("D %s_%03d_%1s ") % (cif_pdb_name, aa_seqres.fasta_pos, \
                                      aa_seqres.amino_acid_short)
        if aa_seqres.pdb_pos == -1 or not aa_seqres.pdb_pos in pos_den_hash:
```

```
            # output count C() = NA
            for index in TAB_PRINT:
                value = value + "C(%d) = NA " % TAB_VALUES[index]
            value = " ?"
        else:
            density_array = pos_den_hash[aa_seqres.pdb_pos]
            for index in TAB_PRINT:
                value = value + "C(%d)= % 3d " % \
                    (TAB_VALUES[index], density_array[index])
            value = value + " %d" % aa_seqres.pdb_pos
        lines.append(value + "\n")
    return lines

################################################################################
# Reads a mmCIF for the given PDB name.
################################################################################
def read_mmcif_file(pdb_name):
    file_name = os.path.join("mmCIF", pdb_name + ".cif")
    if not os.path.exists(file_name):
        print "Error: Couldn't find file:", file_name
        print "Did you forget to run the download_mmCIF.py script?"
        sys.exit(-1)

    input_file_handle = open(file_name, "r")
    input_lines = input_file_handle.readlines()
    input_file_handle.close()

    clean_lines = []
    for line in input_lines:
        clean_lines.append(line.strip())
    return clean_lines

################################################################################
# Get a list of pdb names from the a file.
################################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython calculate_density.py pdb_names.txt\n" \
              "\tor python calculate_density.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])

    return pdb_list

################################################################################
# Gets the output path for the given pdb_name. For example, if the PDB name
# is 1HGXA then the output path would be:
#     density/1HGXA.den
################################################################################
def get_output_path(pdb_name):
    # put the file in a directory named "density"
    if not os.path.exists("density"):
        os.makedirs("density")

    output_path = os.path.join("density", pdb_name + ".den")
    return output_path
```

```
###############################################################################
# Runs the main script.
###############################################################################
def run(pdb_name):
    output_path = get_output_path(pdb_name)
    if os.path.exists(output_path):
        return

    input_lines = read_mmcif_file(pdb_name)
    # The chain name is the last letter of the PDB name in upper case.
    chain_name = pdb_name[-1].upper()

    # Extra data and calculate densities
    aa_seqres_array, alphac_array, cif_pdb_name = \
            extract_all(input_lines, chain_name)
    pos_den_hash = calculate_densities(alphac_array)
    lines = print_densities(cif_pdb_name, aa_seqres_array, pos_den_hash)

    # Save to file
    output_file = open(output_path, "w")
    output_file.write("Number of residues in Sequence = %d\n" %\
                      len(aa_seqres_array))
    output_file.writelines(lines)
    output_file.close()

###############################################################################
# The main function, this gets run first when the program is run from the
# command line.
###############################################################################
if __name__ == "__main__":
    pdb_list = get_pdb_list()

    for index in range(0, len(pdb_list)):
        pdb_name = pdb_list[index]
        # Use lower case name and strip white space.
        pdb_name = pdb_name.lower().strip()

        # Show how much is done
        percent_done = (index + 1.0) / len(pdb_list)
        percent_done = get_int(percent_done * 100.0)
        print index + 1, percent_done, "%", pdb_name

        run(pdb_name)

###############################################################################
# Author: Reecha Nepal
# Date: May 6, 2012
# Purpose: This script downloads a mmCIF file from the wwpdb FTP site.
# To run this script do the following:
#    python download_mmCIF.py pdb_names.txt
#    or
#    python download_mmCIF.py 119LA
# The output files will be created in a directory called "mmCIF".
# File: download_mmCIF.py
###############################################################################

import os
import sys
import gzip
from ftplib import FTP

FTP_ADDRESS = "ftp.wwpdb.org"
FTP_FOLDER = "/pub/pdb/data/structures/divided/mmCIF/"

###############################################################################
# Gets the output path for the given pdb_name. For example, if the PDB name
```

```
# is 1HGXA then the output path would be:
#     mmCIF/1HGXA.cif
###############################################################################
def get_output_path(pdb_name):
    # put the file in a directory named "pdb"
    if not os.path.exists("mmCIF"):
        os.makedirs("mmCIF")

    output_path = os.path.join("mmCIF", pdb_name + ".cif")
    return output_path

###############################################################################
# Download and save the mmCIF file for the given pdb
###############################################################################
def download_mmCIF_for_pdb_name(pdb_name, ftp, output_path):
    # The name of the folder on the FTP site is the 2nd and 3rd character of
    # of the PDB name. For example, for 1r6ja the folder is r6.
    ftp_path = pdb_name[1:3] + "/"

    # The name of the file on the FTP site is the first 4 characters of the
    # PDB name plus the extension ".cif.gz". For example, for 1r6ja, the file
    # name is 1r6j.cif.gz.
    ftp_path = ftp_path + pdb_name[0:4] + ".cif.gz"

    # Start the download.
    zip_path = output_path + ".gz"
    ftp.retrbinary('RETR %s' % ftp_path, open(zip_path, 'wb').write)

    # Unzip the download.
    unzip_file(zip_path)

    # Delete the zip file
    os.remove(zip_path)

###############################################################################
# Unzip a file and save it to disk.
###############################################################################
def unzip_file(in_file_path):
    # If in_file_path is "a/b.cif.gz" then dst_file_path becomes "a/b.cif".
    dst_file_path, file_extension = os.path.splitext(in_file_path)

    src_zip_file = gzip.open(in_file_path, "rb")
    dst_unzip_file = open(dst_file_path, "wb")
    dst_unzip_file.writelines(src_zip_file)
    dst_unzip_file.close()
    src_zip_file.close()

###############################################################################
# Get a list of pdb names from the a file.
###############################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython download_mmCIF.py pdb_names.txt\n" \
              "\tor python download_mmCIF.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])
```

```
    return pdb_list

################################################################################
# Creates a FTP connection
################################################################################
def create_ftp_connection():
    # Login to the FTP site as anonymous
    ftp = FTP(FTP_ADDRESS)
    ftp.login()
    ftp.cwd(FTP_FOLDER)
    return ftp

################################################################################
# Runs the main script.
################################################################################
def run(pdb_name, ftp=None):
    # Use lower case name and strip white space.
    pdb_name = pdb_name.lower().strip()

    output_path = get_output_path(pdb_name)
    if os.path.exists(output_path):
        return

    should_close_ftp = False
    if not ftp:
        ftp = create_ftp_connection()
        should_close_ftp = True

    download_mmCIF_for_pdb_name(pdb_name, ftp, output_path)

    if should_close_ftp:
        ftp.quit()

################################################################################
# The main function, this gets run first when the program is run from the
# command line.
################################################################################
if __name__ == "__main__":
    pdb_list = get_pdb_list()
    ftp = None

    for index in range(0, len(pdb_list)):
        pdb_name = pdb_list[index]

        # Show how much is done
        percent_done = (index + 1.0) / len(pdb_list)
        percent_done = int(percent_done * 100.0)
        print index + 1, percent_done, "%", pdb_name

        if not ftp:
            ftp = create_ftp_connection()
        run(pdb_name, ftp)

    if ftp:
        ftp.quit()

################################################################################
# Author: Reecha Nepal
# Date: May 6, 2012
# Purpose: This script downloads the pdb files for all pdb names in a given
# file. To run this script do the following:
#    python download_pdb.py pdb_names.txt
#    or
#    python download_pdb.py 119LA
# The output files will be created in a directory called "pdb".
# File: download_pdb.py
```

```
##############################################################################

from urllib import urlopen
import os
import time
import sys

##############################################################################
# Download and return the pdb file for the given pdb name.
##############################################################################
def download_pdb_for_pdb_name(pdb_name):
    # The pdb files are stored online with the last letter of the pdb name
    # removed. For example, 119LA becomes 119L.pdb
    pdb_name = pdb_name[:-1]
    url = "http://www.pdb.org/pdb/files/" + pdb_name + ".pdb"
    url_file_handle = urlopen(url)
    pdb_data = url_file_handle.read()
    url_file_handle.close()
    return pdb_data

##############################################################################
# Gets the output path for the given pdb_name. For example, if the PDB name
# is 1HGXA then the output path would be:
#     pdb/1HGXA.pdb
##############################################################################
def get_output_path(pdb_name):
    # put the file in a directory named "pdb"
    if not os.path.exists("pdb"):
        os.makedirs("pdb")

    output_path = os.path.join("pdb", pdb_name + ".pdb")
    return output_path

##############################################################################
# Save the pdb file in a directory called pdb.
##############################################################################
def save_pdb_data(pdb_data, output_path):
    out_file_handle = open(output_path, "wb")
    out_file_handle.write(pdb_data)
    out_file_handle.close()

##############################################################################
# Checks if the pdb data is valid. A valid pdb data should look like this:
#    >HEADER    HYDROLASE(O-GLYCOSYL)                   28-MAY-93   119L
# If there's a server error then we sometimes get data that looks like this:
#    Error:Cannot connect to database
##############################################################################
def pdb_data_is_valid(pdb_data):
    pdb_data_string = pdb_data.decode("utf-8")
    if pdb_data_string[:6] == "HEADER":
        return True
    else:
        return False

##############################################################################
# Get a list of pdb names from a file.
##############################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython download_pdb.py pdb_names.txt\n" \
              "\tor python download_pdb.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
```

```
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])

    return pdb_list

################################################################################
# Runs the main script.
################################################################################
def run(pdb_name):
    output_path = get_output_path(pdb_name)
    if os.path.exists(output_path):
        return

    retry_count = 0
    while True:
        # Download and save the pdb file
        pdb_data = download_pdb_for_pdb_name(pdb_name)
        if pdb_data_is_valid(pdb_data):
            if retry_count > 0:
                print "Downloading pdb data succeeded after", \
                      retry_count, "tries"
            save_pdb_data(pdb_data, output_path)
            break

        # If there was an error then retry up to 8 times.
        if retry_count == 0:
            print "Warning: Downloading pdb data failed, retrying"
        elif retry_count > 8:
            print "Error: Unable to download pdb file for pdb:", \
                    pdb_name, "quitting."
            sys.exit(-1)
        else:
            print "Retrying", retry_count
        retry_count = retry_count + 1

        # Wait 0.1 seconds before trying again incase we're overloading
        # the server.
        time.sleep(0.1)

################################################################################
# The main function, this gets run first when the program is run from the
# command line.
################################################################################
if __name__ == "__main__":
    pdb_list = get_pdb_list()

    for index in range(0, len(pdb_list)):
        pdb_name = pdb_list[index]

        # Show how much is done
        percent_done = (index + 1.0) / len(pdb_list)
        percent_done = int(percent_done * 100.0)
        print index + 1, percent_done, "%", pdb_name

        run(pdb_name)

################################################################################
# Author: Reecha Nepal
# Date: May 6, 2012
# Purpose: Open the output of the naccess program (a .rsa file) and extract
# data from it.
# File: extract_data.py
```

```
################################################################################

#!/usr/bin/python

import os

# Initialize Amino Acid 3-letter to 1-letter associative list
AA_DICTIONARY = {
    'GLY': 'G', 'ALA': 'A', 'VAL': 'V', 'LEU': 'L',
    'ILE': 'I', 'MET': 'M', 'PRO': 'P', 'PHE': 'F',
    'TRP': 'W', 'SER': 'S', 'THR': 'T', 'ASN': 'N',
    'GLN': 'Q', 'TYR': 'Y', 'CYS': 'C', 'LYS': 'K',
    'ARG': 'R', 'HIS': 'H', 'ASP': 'D', 'GLU': 'E'
}

def GetQueryLetter(aa):
    if aa in AA_DICTIONARY:
        return AA_DICTIONARY[aa]
    else:
        return '?'


################################################################
# Reads the .rsa file from the naccess program.
################################################################
def GetNaccessValuesForPDB(pdb_name):
    rsa_file_path = os.path.join("naccess", pdb_name + ".rsa")
    input_file = open(rsa_file_path, "r")
    lines = input_file.readlines()
    input_file.close()

    naccess_table = []
    chain_letter = pdb_name[-1:]

    for line in lines:
        words = line.split()

        if len(words) < 5 or words[0] != "RES":
            continue

        residue = words[1].strip()
        aa = words[2].strip()
        if len(aa) == 1:
            rel = words[5].strip()
        else:
            aa = aa[0]
            rel = words[4].strip()

        if aa != chain_letter:
            continue

        naccess_entry = {}
        naccess_entry["AA"] = residue
        naccess_entry["QueryLetter"] = GetQueryLetter(residue)
        naccess_entry["REL"] = rel
        naccess_entry["CATH"] = aa
        naccess_table.append(naccess_entry)
    return naccess_table


################################################################
#
################################################################
def save_naccess_to_csv(pdb_name, data, output_path):
    f = open(output_path, "w")
    f.write(",AA,REL,CATH\n")
    size = len(data["REL"])
```

```
    for i in range(0, size):
        f.write("%d," % (i+1))
        f.write("%s," % data[i]["AA"])
        f.write("%s," % data[i]["REL"])
        f.write("%s" % data[i]["CATH"])
        f.write("\n")
    f.close()

if __name__ == "__main__":
    pdb_name = "1A4IA"
    data = GetNaccessValuesForPDB(pdb_name)
    save_naccess_to_csv(pdb_name, data, "/Users/reecha/Desktop/a.csv")

##############################################################################
# Author: Reecha Nepal
# Date: May 6, 2012#
# Purpose: This script runs the naccess program.
# To run this script do the following:
#    python run_naccess.py pdb_names.txt
#    or
#    python run_naccess.py 119LA
# The output files will be created in a directory called "naccess".
# The .rsa files in the output folder contain REL values as calculated by
# the naccess program. These REL values are used to compare against predicted
# REL values based on the entropy regression.
# File: run_naccess.py
##############################################################################

import os
import sys
import platform
import tempfile
import shutil

##############################################################################
# Gets the output path for the given pdb_name. For example, if the PDB name
# is 1HGXA then the output path would be:
#     naccess/1HGXA.rsa
##############################################################################
def get_output_path(pdb_name):
    # put the file in a directory named "blast"
    if not os.path.exists("naccess"):
        os.makedirs("naccess")

    output_path = os.path.join("naccess", pdb_name + ".rsa")
    return output_path

##############################################################################
# Get a list of pdb names from the a file.
##############################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython run_naccess.py pdb_names.txt\n" \
              "\tor python run_naccess.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])
```

```
    return pdb_list

################################################################################
# Gets the path to folder that contains the naccess code.
################################################################################
def get_naccess_code_path():
    script_path = os.path.realpath(__file__)
    parent_directory = os.path.dirname(script_path)
    return os.path.join(parent_directory, "NACCESS_Code")

################################################################################
# Gets the path the .pdb file.
################################################################################
def get_pdb_file_path(pdb_name):
    pdb_file_path = os.path.join("pdb", pdb_name + ".pdb")
    if not os.path.exists(pdb_file_path):
        print "Couldn't find pdb file: ", pdb_file_path
        print "Run the download_pdb.py script to download the pdb file first."
        sys.exit(-1)
    return os.path.abspath(pdb_file_path)

################################################################################
# Runs the naccess program
################################################################################
def run_naccess(pdb_name, pdb_file_path, naccess_code_path):
    old_working_directory = os.getcwd()
    temp_dir = tempfile.mkdtemp()
    os.chdir(temp_dir)

    if platform.system() == "Windows":
        naccess_program = os.path.join(naccess_code_path, "naccess_win.exe")
    else:
        naccess_program = os.path.join(naccess_code_path, "naccess")
    vdw_file_path = os.path.join(naccess_code_path, "vdw.radii")
    os.system(naccess_program + " " + pdb_file_path + " -r " + vdw_file_path)

    file_name = pdb_name + '.rsa'
    shutil.copy(file_name, os.path.join(old_working_directory, 'naccess'))
    os.chdir(old_working_directory)
    shutil.rmtree(temp_dir, ignore_errors=True)

################################################################################
# Runs the main script.
################################################################################
def run(pdb_name, naccess_code_path=None):
    if not naccess_code_path:
        naccess_code_path = get_naccess_code_path()
    os.environ["NACCESS_EXE_PATH"] = naccess_code_path

    output_path = get_output_path(pdb_name)
    if os.path.exists(output_path):
        return

    pdb_file_path = get_pdb_file_path(pdb_name)
    run_naccess(pdb_name, pdb_file_path, naccess_code_path)

################################################################################
# The main function, this gets run first when the program is run from the
# command line.
################################################################################
if __name__ == "__main__":
    naccess_code_path = get_naccess_code_path()

    pdb_list = get_pdb_list()
    for index in range(0, len(pdb_list)):
        pdb_name = pdb_list[index]
```

```
        # Show how much is done
        percent_done = (index + 1.0) / len(pdb_list)
        percent_done = int(percent_done * 100.0)
        print index + 1, percent_done, "%", pdb_name

        run(pdb_name, naccess_code_path)

##########################################################################
# Author: Reecha Nepal
# Date: July 23, 2012
# Purpose: Convert blast data to entropy data.
# Usage: import parse_blast
#        data = BlastData()
#        data.ParseQueryAndSubject("nblast_all/1A2KAblast.txt")
#        entropyRecordList = EntropyRecordsForBlastData(blastData)
#
#        At this point entropyRecordList will contain a list of entropy
#        records. Each record contains the entropy value and sequence
#        for a single letter in the blast query sequence.
#
# Revision History
#  v.1.0 11/23/10 Inital version by Reecha Nepal.
#                 This is a python translation of
#                 Radhika-6pointPsiBlastentropy.pl. The original script
#                 was written by D.Chiang and modified by
#                 Radhika Pallavi Mishra. If use6Point is set to False then
#                 this script is equivalent to the bst2entMOD2psiEntropy.pl
#                 perl script.
# File: blast_to_entropy.py
##########################################################################

import parse_blast
import sys
import string
import math

# Specify User Parameters
SCORE_CUT_OFF_PERCENT = 40
HOMOLOG_MIN = 1


#################################################################
# Stores information about a single entropy sequence.
#################################################################
class EntropyRecord:
   def __init__(self):
       self.queryLetter = ""
       self.queryLetterIndex = 0
       self.entropyValue = 0.0
       self.entropySequence = ""


   ################################################################
   # Gets the blast query.
   ################################################################
   def CalculateWithSequence(self, letter, letterIndex, sequence, use6Point):
      self.queryLetter = letter.strip()
      self.queryLetterIndex = letterIndex
      self.entropySequence = sequence
      if use6Point:
         self.entropyValue = Calculate6PointEntropy(self.entropySequence, HOMOLOG_MIN)
      else:
         self.entropyValue = CalculateAllPointEntropy(self.entropySequence, HOMOLOG_MIN)


################################################################
# This function uses the query in the blast data to calculate
```

```
# the entropy for each letter in the blast query sequence.
#################################################################
def EntropyRecordsForBlastData(blastData, use6Point):
   entropyRecordList = []
   query = GetQuery(blastData)

   subjectSequenceList = GetQualifyingSubjectSequenceList(
         query, blastData.recordList)

   # Calculate the entropy for each letter in the query.
   for letterIndex in range(0, len(query)):
      entropySequence = ""
      for subjectSequence in subjectSequenceList:
         entropySequence += subjectSequence[letterIndex]

      record = EntropyRecord()
      record.CalculateWithSequence(query[letterIndex], letterIndex, entropySequence,
use6Point)
      entropyRecordList.append(record)

   return entropyRecordList


#################################################################
# Gets the blast query.
#################################################################
def GetQuery(blastData):
   query = blastData.firstQuerySequence
   # If the blast parser didn't find the query at the top of the
   # blast file then use the query in the first record instead.
   if len(query) == 0 and len(blastData.recordList) > 0:
      query = blastData.recordList[0].querySequence

   # Note that BLAST can substitute 'X' (proteins) or 'N' (nucleotides) into
   # the Query sequence to filter out "low complexity" regions. These
   # residues are kept as X or N in the entropy calculation. However, they
   # can be post-processed when correlated with the PDB information using
   # the residue position number. They are converted to lower case
   # in the output (trick to help merging with pdb2den.pl output,
   # since lowercase sorts after all upper case).
   query = query.replace("X", "x")
   # (Should be removed, N is used for nucleotides only)
   #query = query.replace("N", "n")

   # Extracted all '-' from Query sequence reported from 1st
   # match in BLAST, to take care of case when the 1st match
   # includes insertions (ie the query itself is not found).
   query = query.replace("-", "")

   return query


#################################################################
# Gets the compacted version of the subject sequence.
#################################################################
def GetCompactedSubjectSequence(query, record):
   # The record's query sequence starts at a certain offset from the
   # original query. Fill in the compactQuery with the original query.
   # Fill in the subject sequence with dash characters.
   fillLength = record.queryOffset - 1
   compactQuery = query[0:fillLength] + record.querySequence
   compactSubject = "".ljust(fillLength, '-') + record.subjectSequence

   # Find and delete insertions
   i = 0
   while i < len(compactQuery):
      if compactQuery[i] == "-":
```

```
         compactQuery = compactQuery[:i] + compactQuery[i+1:]
         compactSubject = compactSubject[:i] + compactSubject[i+1:]
      else:
         i += 1

   # If compactQuery is shorter than the query then fill it in with the
   # end of the original query. Fill in the compactSubject with dash
   # characters.
   lengthDiff = len(query) - len(compactSubject)
   if lengthDiff > 0:
      compactQuery = compactQuery + query[-lengthDiff:]
      compactSubject = compactSubject + "".ljust(lengthDiff, "-")

   return compactSubject


##################################################################
# Gets a list of compacted subject sequences.
##################################################################
def GetQualifyingSubjectSequenceList(query, recordList):
   scoreMin = 100
   if len(recordList) > 0:
       scoreMin = recordList[0].scoreBits * SCORE_CUT_OFF_PERCENT / 100.0;

   subjectSequenceList = []
   for record in recordList:
      # If this sequence doesn't qualify, skip
      if record.scoreBits < scoreMin:
         continue
      subjectSequenceList.append(GetCompactedSubjectSequence(query, record))
   return subjectSequenceList


##################################################################
# Calculates entropy from the given sequence. The calculation
# is done by grouping items in the sequence into one of 6
# points.
##################################################################
def Calculate6PointEntropy(sequence, homologMin):
   totalCount = 0
   categoryCount = {"aliphatic" : 0,
                    "aromatic" : 0,
                    "polar" : 0,
                    "positive" : 0,
                    "negative" : 0,
                    "special" : 0}

   for letter in sequence:
      # Ignore any letters that are not upper case
      if letter not in string.ascii_uppercase:
         continue
      totalCount += 1
      if letter in "AVLIMC":
         categoryCount["aliphatic"] += 1
      elif letter in "FWYH":
         categoryCount["aromatic"] += 1
      elif letter in "STNQ":
         categoryCount["polar"] += 1
      elif letter in "KR":
         categoryCount["positive"] += 1
      elif letter in "DE":
         categoryCount["negative"] += 1
      elif letter in "GP":
         categoryCount["special"] += 1

   # If too few homologs then flag as error.
   if totalCount < homologMin:
```

```
      return -1

   entropy = 0.0
   for categoryValue in categoryCount.values():
      if categoryValue > 0:
         prob = float(categoryValue) / totalCount
         entropy = entropy - (prob * (math.log(prob)/math.log(2)))
   return entropy


#################################################################
# Calculates entropy from the given sequence. The calculation
# is done without groping items in the sequence.
#################################################################
def CalculateAllPointEntropy(sequence, homologMin):
   totalCount = 0
   categoryCount = {}

   for letter in sequence:
      # Ignore any letters that are not upper case
      if letter not in string.ascii_uppercase:
         continue
      totalCount += 1
      if letter in categoryCount:
         categoryCount[letter] += 1
      else:
         categoryCount[letter] = 1

   # If too few homologs then flag as error.
   if totalCount < homologMin:
      return -1

   entropy = 0.0
   for categoryValue in categoryCount.values():
      if categoryValue > 0:
         prob = float(categoryValue) / totalCount
         entropy = entropy - (prob * (math.log(prob)/math.log(2)))
   return entropy


#################################################################
# Normally this script is not run directly. Callers should just
# use the EntropyRecordsForBlastData function to get the
# entropy data that they need.
# For debuging purposes though you can call this as follows:
#    python blast_to_entropy.py <pdb_name> <blast_file_name>
#                               <fasta_file_name> <out_file_name>
# This will calculate the entropy and save it in
# <out_file_name>.
#################################################################
if __name__ == "__main__":
   pdb_name = sys.argv[1]
   blast_file_name = sys.argv[2]
   fasta_file_name = sys.argv[3]
   out_file_name = sys.argv[4]

   # Parse the blast file.
   blastData = parse_blast.BlastData()
   blastData.ParseQueryAndSubject(blast_file_name, fasta_file_name)
   entropyRecordList = EntropyRecordsForBlastData(blastData, True)

   out_file = open(out_file_name, "w")
   for r in entropyRecordList:
      # Print the entropy and sequence
      out_file.write("D %s_%03d_%s E = % .3f A= %s\n" %
                     (pdb_name, r.queryLetterIndex + 1, r.queryLetter,
                      r.entropyValue, r.entropySequence))
```

```
   out_file.close()

##############################################################################
# Author: Reecha Nepal
# Date: July 17, 2012
# Purpose: This script prints the bit score for each record in each protein.
# The bit score is the "Score = 780 bits" part in the blast file.
# To run this script do the following:
#    python extract_bit_score.py pdb_names.txt
#    or
#    python extract_bit_score.py 119LA
# File: extract_bit_score.py
##############################################################################

import parse_blast
import sys
import os

##############################################################################
# Get a list of pdb names from the a file.
##############################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython extract_query_length.py pdb_names.txt\n" \
              "\tor python run_all.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])
    return pdb_list

##############################################################################
# Author: Reecha Nepal
# Date: August 13, 2012
# Purpose: Extract data from blast results.
# File: extract_data.py
##############################################################################

#!/usr/bin/python
# File: extract_data.py

import parse_blast
import blast_to_entropy
import fractional_analysis
import parse_density
import sys
import os

##############################################################
#
##############################################################
def save_entropy_to_csv(pdb_name, data, output_path):
    f = open(output_path, "w")
    f.write(",RES,E,E6,FSR,FSHP,AA,REL,CATH\n")
    size = len(data["E"])
    for i in range(0, size):
        f.write("%d," % (i+1))
        f.write("%s," % pdb_name)
```

```
        f.write("%.3f," % data[i]["E"])
        f.write("%.3f," % data[i]["E6"])
        f.write("%f," % data[i]["FSR"])
        f.write("%f," % data[i]["FSHP"])
        f.write("%s," % "NA")
        f.write("%s," % "NA")
        f.write("%s" % "NA")
        f.write("\n")
    f.close()

################################################################
#
################################################################
def save_density_to_csv(pdb_name, data, output_path):
    f = open(output_path, "w")
    f.write(",denB\n")
    size = len(data["denB"])
    for i in range(0, size):
        f.write("%d," % (i+1))
        f.write("%s" % data[i]["denB"])
        f.write("\n")
    f.close()

################################################################
# Calculates the entropy and density values and returns them.
################################################################
def GetEntropyAndDensityValuesForPDB(pdb_name):
    blast_file_path = os.path.join("blast", pdb_name + ".txt")
    fasta_file_path = os.path.join("fasta", pdb_name + ".fasta")
    density_file_path = os.path.join("density", pdb_name + ".den")

    densityRecordList = parse_density.ParseDensityFile(density_file_path)
    blastData = parse_blast.BlastData()
    blastData.ParseQueryAndSubject(blast_file_path, fasta_file_path)
    entropyRecordList = blast_to_entropy.EntropyRecordsForBlastData(
            blastData, False)
    entropyRecordList_6_point = blast_to_entropy.EntropyRecordsForBlastData(
            blastData, True)

    density_table = []
    for record in densityRecordList:
        density_entry = {}
        density_entry["denB"] = record.density_value
        density_entry["QueryLetter"] = record.query_letter
        density_table.append(density_entry)

    entropy_table = []
    for i in range(0, len(entropyRecordList)):
        # Calculate franctions from the entropy record
        f = fractional_analysis.FractionRecord()
        f.CalculateWithEntropyRecord(entropyRecordList[i])
        f6 = fractional_analysis.FractionRecord()
        f6.CalculateWithEntropyRecord(entropyRecordList_6_point[i])

        entropy_entry = {}
        entropy_entry["E"] = entropyRecordList[i].entropyValue
        entropy_entry["E6"] = entropyRecordList_6_point[i].entropyValue
        entropy_entry["QueryLetter"] = entropyRecordList[i].queryLetter
        entropy_entry["FSR"] = f.small_residues_fraction
        entropy_entry["FA"] = f.ala_residue_fraction
        entropy_entry["FG"] = f.gly_residue_fraction
        entropy_entry["FSHP"] = f.strongly_hydrophobic_fraction
        entropy_table.append(entropy_entry)

    return (density_table, entropy_table)

#########################################################################
```

```
# Author: Reecha Nepal
# Date: July 28, 2012
# Purpose: This script prints the query length for each protein. The query is the
# sequence at the top of the blast file.
# To run this script do the following:
#    python extract_query_length.py pdb_names.txt
#    or
#    python extract_query_length.py 119LA
# File: extract_density_frequency.py
################################################################################

import parse_blast
import sys
import os
import exceptions

def get_int(str_value):
    try:
        return int(str_value)
    except exceptions.ValueError:
        return 0

def get_float(str_value):
    try:
        return float(str_value)
    except exceptions.ValueError:
        return 0.0

################################################################################
#
################################################################################
def parse_csv_data(file_path, pdb_name):
    input_file_handle = open(file_path, "r")
    lines = input_file_handle.readlines()
    csv_data = []

    header = None
    for line in lines:
        words = line.strip().split(',')
        if header == None:
            header = words
        elif len(words) > 1:
            record = {}
            for i in range(0, len(words)):
                record[header[i]] = words[i]
            assert record['RES'] == pdb_name
            csv_data.append(record)
    return csv_data

################################################################################
# Get a list of pdb names from the a file.
################################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython extract_density_frequency.py pdb_names.txt\n" \
              "\tor python run_all.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
```

```
    else:
      pdb_list.append(sys.argv[1])
    return pdb_list

if __name__ == "__main__":
    pdb_list = get_pdb_list()
    density_frequence = {}
    max_density = -1
    min_density = 123456

    keys = ["E", "E6", "FSR", "FSHP", "REL"]

    for pdb_name in pdb_list:
      fasta_file_path = os.path.join("csv", pdb_name + ".csv")
      csv_data = parse_csv_data(fasta_file_path, pdb_name)

      for csv_record in csv_data:
        density = get_int(csv_record['denB'])

        if density in density_frequence:
            density_record = density_frequence[density]
        else:
            density_record = {'count':0}

        for key in keys:
            value = get_float(csv_record[key])
            if key in density_record:
                r = density_record[key]
            else:
                r = {'count':0, 'sum':0.0}
            r['count'] = r['count'] + 1
            r['sum'] = r['sum'] + value
            density_record[key] = r
        density_record['count'] = density_record['count'] + 1
        density_frequence[density] = density_record

        if density > max_density:
            max_density = density
        if density < min_density:
            min_density = density

    print "min_density,%d" % min_density
    print "max_density,%d" % max_density

    sys.stdout.write("density,count")
    for key in keys:
        sys.stdout.write(",")
        sys.stdout.write(key)
    sys.stdout.write("\n")

    for i in range(-1, max_density):
        if i in density_frequence:
            density_record = density_frequence[i]
            sys.stdout.write(str(i))
            sys.stdout.write(",")
            sys.stdout.write(str(density_record['count']))
            for key in keys:
                sys.stdout.write(",")
                r = density_record[key]
                average = r['sum'] / r['count']
                sys.stdout.write(str(average))
            sys.stdout.write("\n")
        else:
            sys.stdout.write("%d,0" % i)
            for key in keys:
                sys.stdout.write(",0")
            sys.stdout.write("\n")
```

```
###############################################################################
# Author: Reecha Nepal
# Date: July 17, 2012
# Purpose: This script prints the query length for each protein. The query is
# the sequence at the top of the blast file.
# To run this script do the following:
#    python extract_query_length.py pdb_names.txt
#    or
#    python extract_query_length.py 119LA
# File: extract_query_length.py
###############################################################################

import parse_blast
import sys
import os

###############################################################################
# Get a list of pdb names from the a file.
###############################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython extract_query_length.py pdb_names.txt\n" \
              "\tor python run_all.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])
    return pdb_list

if __name__ == "__main__":
    pdb_list = get_pdb_list()
    for pdb_name in pdb_list:
      fasta_file_path = os.path.join("fasta", pdb_name + ".fasta")
      query = parse_blast.ParseQueryFromFastaFile(fasta_file_path)
      print "%s,%s" % (pdb_name, len(query))

###############################################################################
# Author: Reecha Nepal
# Date: July 17, 2012
# Purpose: This script prints the record length for each protein.
# sequence at the top of the blast file.
# To run this script do the following:
#    python extract_record_length.py pdb_names.txt
#    or
#    python extract_record_length.py 119LA
# File: extract_record_length.py
###############################################################################

import parse_blast
import sys
import os

###############################################################################
# Get a list of pdb names from the a file.
###############################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
```

```
        print "To run this script do the following:\n" \
              "\tpython extract_query_length.py pdb_names.txt\n" \
              "\tor python run_all.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])
    return pdb_list

if __name__ == "__main__":
    pdb_list = get_pdb_list()
    for pdb_name in pdb_list:
      blast_file_path = os.path.join("blast", pdb_name + ".txt")
      fasta_file_path = os.path.join("fasta", pdb_name + ".fasta")
      blastData = parse_blast.BlastData()
      blastData.ParseQueryAndSubject(blast_file_path, fasta_file_path)
      print "%s,%s" % (pdb_name, len(blastData.recordList))

###############################################################################
# Author: Reecha Nepal
# Date: July 28, 2012
# Purpose: This script prints the query length for each protein. The query
# is the sequence at the top of the blast file.
# To run this script do the following:
#    python extract_query_length.py pdb_names.txt
#    or
#    python extract_query_length.py 119LA
# File: extract_rel_frequency.py
###############################################################################

import math
import parse_blast
import sys
import os
import exceptions

def get_int(str_value):
    try:
        return int(str_value)
    except exceptions.ValueError:
        return 0

def get_float(str_value):
    try:
        return float(str_value)
    except exceptions.ValueError:
        return 0.0

###############################################################################
#
###############################################################################
def parse_csv_data(file_path, pdb_name):
    input_file_handle = open(file_path, "r")
    lines = input_file_handle.readlines()
    csv_data = []

    header = None
    for line in lines:
        words = line.strip().split(',')
```

```
        if header == None:
            header = words
        elif len(words) > 1:
            record = {}
            for i in range(0, len(words)):
                record[header[i]] = words[i]
            assert record['RES'] == pdb_name
            csv_data.append(record)
    return csv_data

################################################################################
# Get a list of pdb names from the a file.
################################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython extract_rel_frequency.py pdb_names.txt\n" \
              "\tor python run_all.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name)
    else:
      pdb_list.append(sys.argv[1])
    return pdb_list

################################################################################
#
################################################################################

def get_bucket(rel):
    if rel <= 0:
        return 1
    a = rel / 10.0
    return int(math.ceil(a) + 1.0)


def get_bucket_range(bucket):
    if bucket == 1:
        return "0"
    else:
        rel = (bucket - 1) * 10
        return str(rel - 10) + " < rel <= " + str(rel)

if __name__ == "__main__":
    pdb_list = get_pdb_list()
    rel_frequence = {}
    max_rel = -1
    min_rel = 1234567

    keys = ["denB", "E", "E6", "FSR", "FSHP", "REL"]

    for pdb_name in pdb_list:
      fasta_file_path = os.path.join("csv", pdb_name + ".csv")
      csv_data = parse_csv_data(fasta_file_path, pdb_name)

      for csv_record in csv_data:
        rel = get_float(csv_record['REL'])
        rel_bucket = get_bucket(rel)

        if rel_bucket in rel_frequence:
```

```
            rel_record = rel_frequence[rel_bucket]
        else:
            rel_record = {'count':0}

        for key in keys:
            value = get_float(csv_record[key])
            if key in rel_record:
                r = rel_record[key]
            else:
                r = {'count':0, 'sum':0.0}
            r['count'] = r['count'] + 1
            r['sum'] = r['sum'] + value
            rel_record[key] = r
        rel_record['count'] = rel_record['count'] + 1
        rel_frequence[rel_bucket] = rel_record

        if rel > max_rel:
            max_rel = rel
        if rel < min_rel:
            min_rel = rel

    print "min_rel,%d" % min_rel
    print "max_rel,%d" % max_rel

    sys.stdout.write("REL bucket,REL range,count")
    for key in keys:
        sys.stdout.write(",")
        sys.stdout.write(key)
    sys.stdout.write("\n")

    max_rel_bucket = get_bucket(max_rel)
    for i in range(1, max_rel_bucket):
        sys.stdout.write(str(i))
        sys.stdout.write(",")
        sys.stdout.write(get_bucket_range(i))
        sys.stdout.write(",")
        if i in rel_frequence:
            rel_record = rel_frequence[i]
            sys.stdout.write(str(rel_record['count']))
            for key in keys:
                sys.stdout.write(",")
                r = rel_record[key]
                average = r['sum'] / r['count']
                sys.stdout.write(str(average))
            sys.stdout.write("\n")
        else:
            sys.stdout.write("0")
            for key in keys:
                sys.stdout.write(",0")
            sys.stdout.write("\n")

##############################################################################
# Author: Reecha Nepal
# Date: August 13, 2012
# Purpose: Calculate fraction record for blast data.
# Usage: import parse_blast
#        import blast_to_entropy
#        data = parse_blast.BlastData()
#        data.ParseQueryAndSubject("nblast_all/1A2KAblast.txt")
#        entropyRecordList = EntropyRecordsForBlastData(blastData)
#        fractionRecord = FractionRecord()
#        fractionRecord.CalculateWithEntropyRecord(entropyRecordList[0])
#
#        At this point fractionRecord will contain fraction values for the
#        first entropy record.
#
# Revision History
```

```
#  v.1.0 11/23/10 Inital version by Reecha Nepal.
#                 This is a python translation of
#                 extract_fractanalysis_entropy_aggr.pl. The original script
#                 was written by Radhika Pallavi Mishra.
# File: fractional_analysis.py
################################################################################

import parse_blast
import blast_to_entropy
import sys


#################################################################
# Stores fractions computed from a single entropy record.
#################################################################
class FractionRecord:
   def __init__(self):
       self.gap_fraction = 0.0
       self.small_residues_fraction = 0.0
       self.ala_residue_fraction = 0.0
       self.gly_residue_fraction = 0.0
       self.strongly_hydrophobic_fraction = 0.0
       self.non_strongly_hydrophobic_fraction = 0.0

   ################################################################
   # Fills in the FractionRecord object using fractions computed
   # from the given entropy record.
   ################################################################
   def CalculateWithEntropyRecord(self, entropyRecord):
      gap_count = 0
      small_residues_count = 0
      ala_residue_count = 0
      gly_residue_count = 0
      strongly_hydrophobic_count = 0
      total_length = len(entropyRecord.entropySequence)
      for letter in entropyRecord.entropySequence:
         if letter in "-":
            gap_count += 1
         elif letter in "AG":
            small_residues_count += 1
            if letter == "A":
                ala_residue_count += 1
            else:
                gly_residue_count += 1
         elif letter in "VILFYMW":
            strongly_hydrophobic_count += 1

      num_non_gap_amino_acids = total_length - gap_count
      if num_non_gap_amino_acids > 0:
         self.gap_fraction = float(gap_count) / num_non_gap_amino_acids
         self.small_residues_fraction = float(small_residues_count) /
num_non_gap_amino_acids
         self.ala_residue_fraction = float(ala_residue_count) / num_non_gap_amino_acids
         self.gly_residue_fraction = float(gly_residue_count) / num_non_gap_amino_acids
         self.strongly_hydrophobic_fraction = float(strongly_hydrophobic_count) /
num_non_gap_amino_acids
         self.non_strongly_hydrophobic_fraction = 1.0 -
self.strongly_hydrophobic_fraction
      else:
         self.gap_fraction = 0.0
         self.small_residues_fraction = 0.0
         self.ala_residue_fraction = 0.0
         self.gly_residue_fraction = 0.0
         self.strongly_hydrophobic_fraction = 0.0
         self.non_strongly_hydrophobic_fraction = 0.0
```

```
#################################################################
# Normally this script is not run directly. Callers should just
# use the FractionRecord class to get the fraction data that
# they need.
# For debuging purposes though you can call this as follows:
#    python fractional_analysis.py <pdb_name> <blast_file_name> <out_file_name>
# This will compute fraction values for each letter in the
# blast query and save the result to <out_file_name>.
#################################################################
if __name__ == "__main__":
   pdb_name = sys.argv[1]
   blast_file_name = sys.argv[2]
   out_file_name = sys.argv[3]

   # Parse the blast file.
   blastData = parse_blast.BlastData()
   blastData.ParseQueryAndSubject(blast_file_name)
   entropyRecordList = blast_to_entropy.EntropyRecordsForBlastData(blastData, False)

   out_file = open(out_file_name, "w")
   for entropyRecord in entropyRecordList:
      f = FractionRecord()
      f.CalculateWithEntropyRecord(entropyRecord)
      # Print the entropy and fractions
      out_file.write("E=%.3f,FG=%f,FSR=%f,FSHP=%f,FNSHP=%f\n" %
                     (entropyRecord.entropyValue,
                      f.gap_fraction,
                      f.small_residues_fraction,
                      f.strongly_hydrophobic_fraction,
                      f.non_strongly_hydrophobic_fraction))
   out_file.close()

#################################################################################
# Author: Reecha Nepal
# Date: July 17, 2012
# Purpose: Parse blast file data.
# Usage: data = BlastData()
#        data.ParseQueryAndSubject("nblast_all/1A2KAblast.txt",
#                                  "fasta/1A2kA.fasta")
#
#        At this point data.firstQuerySequence will contain the
#        blast query. For example "MGDKPIWEQ..."
#        The query subject and records are stored in data.recordList.
#
# Revision History
#  v.1.0 11/23/10 Inital version by Reecha Nepal.
#                 This is a python translation of
#                 Radhika-6pointPsiBlastentropy.pl. The original script
#                 was written by D.Chiang and modified by
#                 Radhika Pallavi Mishra.
# File: parse_blast.py
#################################################################################

import sys

#################################################################
# Stores a single query and subject record.
#################################################################
class QueryAndSubjectRecord:
   def __init__(self):
      self.scoreBits = 0.0
      self.percentIdentities = 0.0
      self.percentPositives = 0.0
      self.expectValue = 0
      self.querySequence = ""
      self.subjectSequence = ""
      self.queryOffset = 0
```

```
      self.lineStartIndex = 0

   ###############################################################
   # Fills in the QueryAndSubjectRecord object by parsing data
   # in the given lines.
   ###############################################################
   def ParseLines(self, lines, lineStartIndex):
      # First line should look like this
      # Score =  187 bits (475),  Expect = 4e-46, Method: Composition-based stats.
      words = lines[0].split()
      self.scoreBits = ParseFloat(words[2])
      self.expectValue = ParseFloat(words[7])
      self.lineStartIndex = lineStartIndex

      # Second line should look like this:
      # Identities = 122/127 (97%), Positives = 126/127 (99%), Gaps = 0/127 (0%)
      words = lines[1].split()
      self.percentIdentities = ParseFloat(words[3]) / 100.0
      self.percentPositives = ParseFloat(words[7]) / 100.0

      # Read the query sequence. The lines look like this:
      # Query  121  LALHNFG  127
      for line in lines:
         if line[:5] == "Query":
            words = line.split()
            self.querySequence = self.querySequence + words[2]
            if self.queryOffset == 0:
               self.queryOffset = int(words[1])

      # Read the subject sequence. The lines look like this:
      # Sbjct  121  LALHNFG  127
      for line in lines:
         if line[:5] == "Sbjct":
            words = line.split()
            self.subjectSequence = self.subjectSequence + words[2]


###################################################################
# This class stores data from the results of a blast query.
# It stores the original query and a list of query and subject
# records.
###################################################################
class BlastData:
   def __init__(self):
      self.recordList = []
      self.firstQuerySequence = ""


   ###############################################################
   # Fills in the BlastData object by parsing the data in the
   # given blast file.
   ###############################################################
   def ParseQueryAndSubject(self, blast_file_name, fasta_file_name):
      self.firstQuerySequence = ParseQueryFromFastaFile(fasta_file_name)

      input_file = open(blast_file_name, "r")
      lines = input_file.readlines()
      input_file.close()

      # Parse the subject and query records.
      lineIndex = 0
      while True:
         (startIndex, endIndex) = FindNextQueryAndSubjectLines(lines, lineIndex)
         if startIndex == -1:
            break
         record = QueryAndSubjectRecord()
         record.ParseLines(lines[startIndex:endIndex+1], startIndex)
```

```
         self.recordList.append(record)
         lineIndex = endIndex + 1

################################################################
# Reads a fast file and returns the query sequence from it.
################################################################
def ParseQueryFromFastaFile(fasta_file_name):
   input_file = open(fasta_file_name, "r")
   lines = input_file.readlines()
   input_file.close()

   query = ""
   for line in lines:
      if line[:1] == '>':
         continue;
      query = query + line.strip()
   return query

################################################################
# Change values from string to float.
################################################################
def ParseFloat(floatString):
   # If the string is "e-10" then change it to "1e-10". Otherwise python
   # won't be able to parse it.
   if floatString[:1] == "e":
      floatString = "1" + floatString
   # Remove any trailing commas and brackets and percent signs
   floatString = floatString.strip(",()%")
   return float(floatString)


################################################################
# Finds the next Query and Subject record in the given lines.
# Returns a tuple with the start and end line indexes for the
# record. If no record is found then it returns (-1, -1).
################################################################
def FindNextQueryAndSubjectLines(lines, startIndex):
   recordStartIndex = 0
   recordEndIndex = 0

   # Find the start of the next record
   for lineIndex in range(startIndex, len(lines)):
      line = lines[lineIndex]
      if line.find("Score = ") != -1:
         break
   if lineIndex >= len(lines) - 1:
      return (-1, -1)
   else:
      recordStartIndex = lineIndex

   # Find the end of the next record
   blankLineCount = 0
   for lineIndex in range(recordStartIndex, len(lines)):
      line = lines[lineIndex].strip()
      if len(line) == 0:
         blankLineCount += 1
      else:
         blankLineCount = 0
      if blankLineCount == 2:
         break
   if lineIndex >= len(lines) - 1:
      return (-1, -1)
   else:
      recordEndIndex = lineIndex

   return (recordStartIndex, recordEndIndex-2)
```

```
################################################################
# Normally this script is not run directly. Callers should just
# use the BlastData object to get the data they need.
# For debuging purposes though you can call this as follows:
#    python parse_blast.py <blast_file_name> <fasta_file_name>
# This will print the first query sequence, the first record,
# and the total number of records.
################################################################
if __name__ == "__main__":
   blast_file_name = sys.argv[1]
   fasta_file_name = sys.argv[2]

   data = BlastData()
   data.ParseQueryAndSubject(blast_file_name, fasta_file_name)
   print "First query sequence is", data.firstQuerySequence
   print "found", len(data.recordList), "records"
   record = data.recordList[0]
   print "The first record is"
   print "scoreBits", record.scoreBits
   print "percentIdentities", record.percentIdentities
   print "percentPositives", record.percentPositives
   print "expectValue", record.expectValue
   print "querySequence", record.querySequence
   print "subjectSequence", record.subjectSequence
   print "queryOffset", record.queryOffset

################################################################################
# Author: Reecha Nepal
# Date: May 6, 2012
# "Parse Density"
#
# Purpose: Parse the density file.
# Usage: import parse_density
#        densityRecordList = parse_density.ParseDensityFile("1a1ia.den")
#        At this point densityRecordList will contain a list of density values.
#
# Revision History
#  v.1.0 11/23/10 Inital version by Reecha Nepal.
#                 This is a python translation of
#                 svm_extract_fractentropy_density_aggr.py. The original script
#                 was written by Radhika Pallavi Mishra.
# File: parse_density.py
################################################################################

import sys


################################################################
# Stores a single density value and the pdb position for that
# value.
################################################################
class DensityRecord:
   def __init__(self):
       density_value = ""
       pdb_pos = ""
       query_letter = ""

   ################################################################
   # Fills in the DensityRecord object by parsing data in the
   # given line.
   ################################################################
   def ParseLine(self, line):
      words = line.split()
      self.density_value = words[9].strip()
      self.pdb_pos = words[16].strip()
      query_words = words[1].split('_')
```

```
      self.query_letter = '?'
      if len(query_words) == 3:
        query_word = query_words[2].strip().upper()
        if len(query_word) == 1:
            self.query_letter = query_word

##############################################################
# Parses the given density file and returns a list of density
# records.
##############################################################
def ParseDensityFile(density_file_name):
   density_file = open(density_file_name, "r")
   lines = density_file.readlines()
   density_file.close()

   densityRecordList = []
   for line in lines:
      if line.find("C(9)") != -1:
         densityRecord = DensityRecord()
         densityRecord.ParseLine(line)
         densityRecordList.append(densityRecord)
   return densityRecordList


##################################################################
# Normally this script is not run directly. Callers should just
# use the DensityRecord object to get the data they need.
# For debuging purposes though you can call this as follows:
#    python parse_density.py <density_file_name>
# This will print the number of density records, and the
# density value and pdb position of the first density record.
##################################################################
if __name__ == "__main__":
   density_file_name = sys.argv[1]

   densityRecordList = ParseDensityFile(density_file_name)
   print "found", len(densityRecordList), "records"
   record = densityRecordList[10]
   print "The first record is"
   print "density_value", record.density_value
   print "pdb_pos", record.pdb_pos

#############################################################################
# Author: Reecha Nepal
# Date: November 23, 2010
# Purpose: This script prints the density and entropy values for all density and
# blast files in the given directory.
# Usage: python svm_extract_fractentropy_density_aggr.py <density_directory>
#   <blast_directory> <out_file_name> <6Point|AllPoint>
# The last parameter to this script should either be 6Point or AllPoint.
# If 6Point is specified then the entropy value will be calculated
# by grouping items in the sequence into one of six categories.
# If AllPoint is specified then items in the sequence will not be
# grouped when calculating the entropy.
#
# Revision History
#  v.1.0 11/23/10 Inital version by Reecha Nepal.
#                 This is a python translation of
#                 svm_extract_fractentropy_density_aggr.py. The original script
#                 was written by Radhika Pallavi Mishra.
# File: svm_extract_fractentropy_density_aggr.py
#############################################################################

import parse_blast
import blast_to_entropy
import fractional_analysis
import parse_density
```

```
import sys
import os


#############################################################
# Prints density and entropy fraction values for the given
# blast file and density file.
#############################################################
def PrintDensityAndEntropyValue(out_file, pdb_name, density_file_name, blast_file_name,
use6Point):
   # Read the density data
   densityRecordList = parse_density.ParseDensityFile(density_file_name)

   # Read the entropy data
   blastData = parse_blast.BlastData()
   blastData.ParseQueryAndSubject(blast_file_name)
   entropyRecordList = blast_to_entropy.EntropyRecordsForBlastData(blastData, use6Point)

   # TODO The density record list should be the same size as the entropy
   # record list. Unfortunately there are some bugs in the script that creates
   # the density record list so the two lists are not always the same size.
   # Until this is fixed just ignore the extra data.
   recordCount = len(densityRecordList)
   if recordCount > len(entropyRecordList):
      recordCount = len(entropyRecordList)

   # Print a line for each density
   for i in range(0, recordCount):
      densityRecord = densityRecordList[i]
      entropyRecord = entropyRecordList[i]

      # Calculate franctions from the entropy record
      f = fractional_analysis.FractionRecord()
      f.CalculateWithEntropyRecord(entropyRecord)

      # Print everything

out_file.write("ProtName=%s,PDBPos=%s,Den=%s,E=%.3f,FG=%f,FSR=%f,FSHP=%f,FNSHP=%f\n" %
                     (pdb_name,
                      densityRecord.pdb_pos,
                      densityRecord.density_value,
                      entropyRecord.entropyValue,
                      f.gap_fraction,
                      f.small_residues_fraction,
                      f.strongly_hydrophobic_fraction,
                      f.non_strongly_hydrophobic_fraction));


#############################################################
# Prints density and entropy fraction values for all density
# and blast files in the given directories.
#############################################################
def PrintDensityAndEntropyForDirectory(out_file, density_directory, blast_directory,
use6Point):
   density_file_list = os.listdir(density_directory)
   blast_file_list = os.listdir(blast_directory)

   for index in range(0, len(density_file_list)):
      file_name = density_file_list[index]
      pdb_name = file_name[0:5].upper()
      blast_file_name = FindFileWithPDBNameInDirectory(blast_directory, blast_file_list,
pdb_name)
      if len(blast_file_name) == 0:
         continue

      print index + 1, "of", len(density_file_list), pdb_name
```

```
      density_file_name = os.path.join(density_directory, file_name)
      PrintDensityAndEntropyValue(out_file, pdb_name, density_file_name, blast_file_name,
use6Point)


############################################################
# Finds a file in the given directory that starts with the
# pdb name.
############################################################
def FindFileWithPDBNameInDirectory(directory_name, direstory_file_list, pdb_name):
   pdb_name = pdb_name.upper()
   pdb_len = len(pdb_name)
   for file_name in direstory_file_list:
      name_prefix = file_name[0:pdb_len].upper()
      if name_prefix == pdb_name:
         return os.path.join(directory_name, file_name)
   return ""

############################################################
# This script prints the density and entropy values for all
# density and blast files in the given directory. To use
# this script call it as follows:
#     python svm_extract_fractentropy_density_aggr.py <density_directory>
<blast_directory> <out_file_name> <6Point|AllPoint>
############################################################
if __name__ == "__main__":
   density_directory = sys.argv[1]
   blast_directory = sys.argv[2]
   out_file_name = sys.argv[3]
   entropyGroupType = sys.argv[4]
   use6Point = entropyGroupType == "6Point"

   out_file = open(out_file_name, "w")
   PrintDensityAndEntropyForDirectory(out_file, density_directory, blast_directory,
use6Point)
   out_file.close()
```

```
################################################################################
# Author: Reecha Nepal
# Date: July 24, 2012
# Purpose: Align two tables based on QueryLetter.
# File: align_tables.py
################################################################################

from difflib import SequenceMatcher
import copy

def get_sequence_from_table(table):
    result = ''
    index = 0
    for entry in table:
        assert len(entry['QueryLetter']) == 1
        result = result + entry['QueryLetter']
        index = index + 1
    return result

def join_dictionaries(dict1, dict2):
    result = {}
    if dict1:
        result = copy.deepcopy(dict1)
    if dict2:
        tmp_dict = copy.deepcopy(dict2)
        for key in tmp_dict.keys():
            result[key] = tmp_dict[key]
    return result

def get_nil_entry(table):
```

```
    if len(table) > 0:
        result = {}
        entry = table[0]
        for key in entry.keys():
            result[key] = '-1'
        return result
    else:
        return {}

def get_aligned_index(naccess_sequence, naccess_index,
                      entropy_sequence, entropy_index,
                      sequence_matcher):
    if len(naccess_sequence) <= naccess_index:
        return -1
    if len(entropy_sequence) <= entropy_index:
        return -1
    if naccess_sequence[naccess_index] == '?':
        return -1
    if naccess_sequence[naccess_index] == entropy_sequence[entropy_index]:
        return entropy_index

    match_naccess, match_entropy, match_len = sequence_matcher.find_longest_match(
        naccess_index, len(naccess_sequence),
        entropy_index, len(entropy_sequence))

    for cur_entropy_index in range(entropy_index,  match_entropy + 1):
        if naccess_sequence[naccess_index] == entropy_sequence[cur_entropy_index]:
            return cur_entropy_index
    return -1

def align_entropy_to_table(entropy_table, naccess_table):
    entropy_sequence = get_sequence_from_table(entropy_table)
    naccess_sequence = get_sequence_from_table(naccess_table)
    sequence_matcher = SequenceMatcher(None, naccess_sequence, entropy_sequence, False)

    aligned_table = []
    entropy_index = 0
    for naccess_index in range(0, len(naccess_sequence)):
        new_entropy_index = get_aligned_index(naccess_sequence, naccess_index,
                                              entropy_sequence, entropy_index,
                                              sequence_matcher)
        if new_entropy_index == -1:
            continue

        aligned_table.append(join_dictionaries(entropy_table[new_entropy_index],
                                               naccess_table[naccess_index]))
        entropy_index = new_entropy_index + 1
    return aligned_table

def align_density_to_table(density_table, naccess_table):
    density_sequence = get_sequence_from_table(density_table)
    naccess_sequence = get_sequence_from_table(naccess_table)
    sequence_matcher = SequenceMatcher(None, naccess_sequence, density_sequence, False)

    nil_density_entry = get_nil_entry(density_table)

    aligned_table = []
    block_index = 0
    matching_blocks = sequence_matcher.get_matching_blocks()
    match_naccess, match_density, match_len = matching_blocks[block_index]

    for naccess_index in range(0, len(naccess_sequence)):
        if (    naccess_index >= (match_naccess + match_len) and
                block_index < (len(matching_blocks) - 1)):
            block_index = block_index + 1
            match_naccess, match_density, match_len = matching_blocks[block_index]
```

```
        if naccess_index < match_naccess:
            aligned_table.append(join_dictionaries(nil_density_entry,
                                                   naccess_table[naccess_index]))
        elif naccess_index >= (match_naccess + match_len):
            aligned_table.append(join_dictionaries(nil_density_entry,
                                                   naccess_table[naccess_index]))
        else:
            delta = naccess_index - match_naccess
            density_index = match_density + delta
            aligned_table.append(join_dictionaries(density_table[density_index],
                                                   naccess_table[naccess_index]))
    return aligned_table

def align_tables(density_table, entropy_table, naccess_table):
    result = align_entropy_to_table(entropy_table, naccess_table)
    result = align_density_to_table(density_table, result)
    return result

################################################################################
# Author: Reecha Nepal
# Date: May 6, 2012
# Purpose: Combine all .csv files in the current directory.
# File: combine_csv_files.py
################################################################################

import os
import sys

header = None
output_lines = []

files = os.listdir(".")
for file in files:
    words = file.split(".")
    if len(words) != 2 or words[1] != "csv":
        continue
    file = open(file, "r")
    input_lines = file.readlines()
    file.close()

    if len(input_lines) == 0:
        print "file", file, "is empty"
        continue

    if not header:
        header = input_lines[0].strip()
        output_lines.append(header + "\n")

    for index in range(1, len(input_lines)):
        line = input_lines[index].strip()
        output_lines.append(line + "\n")

output_file = open("combined.csv", "w")
output_file.writelines(output_lines)
output_file.close()

################################################################################
# Author: Reecha Nepal
# Date: April 16, 2012
# Purpose: This program calculates the relative solvent accessibility (REL)
# using the following models such as:
#   REL ~ E20 + E6 + FSR + FSHP + as.factor(AA)
# The coefficient for the above model are caclulated using training data
# (Brel2Data268.csv). This model is then applied to the experimental data
# (Brel2Data215.csv or other files) to predict new REL values.
#
# The predicted REL values and the actual REL values are then converted to
```

```
# binary using the following forumal:
#   if (binary_actual_REL >= 20)
#     binary_actual_REL = 1
#   else
#     binary_actual_REL = 0
#
#   if (binary_predicted_REL >= 20)
#     binary_predicted_REL = 1
#   else
#     binary_predicted_REL = 0
#
# The accuracy of the model is then calculated by comparing
# binary_predicted_REL with binary_actual_REL.
#
# To run this program do the following:
# > R -f accuracy_v2.R <folder for training data> <folder for experimental data>
# for example:
# > R -f accuracy_v2.R --args 268_csv_folder 215_csv_folder
#
# File:accuracy.R
################################################################################

################################################################################
# Reads all the CSV files in the given folder and creates a single data frame.
################################################################################
read_data_from_files <- function(folder_name) {
    file_list = list.files(folder_name, pattern="*.csv", full.names=TRUE)
    E20 = c()
    E6 = c()
    FSR = c()
    #FA = c()
    #FG = c()
    FSHP = c()
    AA = c()
    REL = c()
    RES = c()
    for (file in file_list) {
        data = read.csv(file, sep = ",", header = TRUE)
        E20 = c(E20, data$E)
        E6 = c(E6, data$E6)
        FSR = c(FSR, data$FSR)
        #FA = c(FA, data$FA)
        #FG = c(FG, data$FG)
        FSHP = c(FSHP, data$FSHP)
        AA = c(AA, as.vector(data$AA))
        REL = c(REL, data$REL)
        RES = append(RES, as.vector(data$RES))
    }

    return (data.frame(E20 = E20,
                       E6 = E6,
                       FSR = FSR,
                       #FA = FA,
                       #FG = FG,
                       FSHP = FSHP,
                       AA = AA,
                       REL = REL,
                       RES = RES))
}


################################################################################
# Applies the given model the experimental data. Returns the number of REL
# values where the predicted value matches the actual values.
# Use the following formula:
#     Predicted REL >  23
#     NACCESS REL   >= 20
```

```
##############################################################################
apply_model <- function(model, exp_data) {
    match_count = 0
    predicted_values = predict(model, exp_data)
    for (i in 1:length(predicted_values)) {
        actual_REL = exp_data$REL[i]
        if (actual_REL >= 20)
            actual_REL_Binary = 1
        else
            actual_REL_Binary = 0

        predicted_REL = predicted_values[i]
        if (predicted_REL > 23) # 25.2
            predicted_REL_Binary = 1
        else
            predicted_REL_Binary = 0

        if (actual_REL_Binary == predicted_REL_Binary)
            match_count = match_count + 1
    }
    return(match_count)
}


##############################################################################
# Save the predicted values in a CVS file for each PDB name.
##############################################################################
save_predicted_values <- function(i, model, exp_data) {
    folder_name = paste("predicted_", i, sep="")
    dir.create(folder_name, showWarnings=FALSE)

    predicted_values = predict(model, exp_data)
    for (i in 1:length(predicted_values)) {
        pdb_name = exp_data$RES[i]
        # remove any white space
        pdb_name = gsub(" +", "", pdb_name)
        file_name = paste(folder_name, "/", pdb_name, ".txt", sep="")
        cat(predicted_values[i], file=file_name, sep="\n", append=TRUE)
    }
}


##############################################################################
# Read all the training csv files and setup models based on the data.
##############################################################################
args = commandArgs(trailingOnly = TRUE)
training_data = read_data_from_files(args[1])

models = list()
model_names = c()
models[[1]] = lm(REL ~ E20, training_data)
model_names[1] = "REL ~ E20"
models[[2]] = lm(REL ~ E6, training_data)
model_names[2] = "REL ~ E6"
models[[3]] = lm(REL ~ FSHP, training_data)
model_names[3] = "REL ~ FSHP"
models[[4]] = lm(REL ~ FSHP + as.factor(AA), training_data)
model_names[4] = "REL ~ FSHP + as.factor(AA)"
models[[5]] = lm(REL ~ AA, training_data)
model_names[5] = "REL ~ AA"
models[[6]] = lm(REL ~ E20 + as.factor(AA), training_data)
model_names[6] = "REL ~ E20 + as.factor(AA)"
models[[7]] = lm(REL ~ E6 + as.factor(AA), training_data)
model_names[7] = "REL ~ E6 + as.factor(AA)"
models[[8]] = lm(REL ~ E20 + E6 + as.factor(AA), training_data)
model_names[8] = "REL ~ E20 + E6 + as.factor(AA)"
models[[9]] = lm(REL ~ E20 + E6, training_data)
```

```
model_names[9] = "REL ~ E20 + E6"
models[[10]] = lm(REL ~ E20 + FSR + FSHP + as.factor(AA), training_data)
model_names[10] = "REL ~ E20 + FSR + FSHP + as.factor(AA)"
models[[11]]= lm(REL ~ E6 + FSR + FSHP + as.factor(AA), training_data)
model_names[11] = "REL ~ E6 + FSR + FSHP + as.factor(AA)"
models[[12]] = lm(REL ~ E20 + E6 + FSR + FSHP + as.factor(AA), training_data)
model_names[12] = "REL ~ E20 + E6 + FSR + FSHP + as.factor(AA)"

#models[[13]] = lm(REL ~ E20 + FA + FG + FSHP + as.factor(AA), training_data)
#model_names[13] = "REL ~ E20 + FA + FG + FSHP + as.factor(AA)"
#models[[14]] = lm(REL ~ E20 + FA + FSHP + as.factor(AA), training_data)
#model_names[14] = "REL ~ E20 + FA + FSHP + as.factor(AA)"
#models[[15]] = lm(REL ~ E20 + FG + FSHP + as.factor(AA), training_data)
#model_names[15] = "REL ~ E20 + FG + FSHP + as.factor(AA)"

#models[[16]]= lm(REL ~ E6 + FA + FG + FSHP + as.factor(AA), training_data)
#model_names[16] = "REL ~ E6 + FA + FG + FSHP + as.factor(AA)"
#models[[17]]= lm(REL ~ E6 + FA + FSHP + as.factor(AA), training_data)
#model_names[17] = "REL ~ E6 + FA + FSHP + as.factor(AA)"
#models[[18]]= lm(REL ~ E6 + FG + FSHP + as.factor(AA), training_data)
#model_names[18] = "REL ~ E6 + FG + FSHP + as.factor(AA)"

#models[[19]] = lm(REL ~ E20 + E6 + FA + FG + FSHP + as.factor(AA), training_data)
#model_names[19] = "REL ~ E20 + E6 + FA + FG + FSHP + as.factor(AA)"
#models[[20]] = lm(REL ~ E20 + E6 + FA + FSHP + as.factor(AA), training_data)
#model_names[20] = "REL ~ E20 + E6 + FA + FSHP + as.factor(AA)"
#models[[21]] = lm(REL ~ E20 + E6 + FG + FSHP + as.factor(AA), training_data)
#model_names[21] = "REL ~ E20 + E6 + FG + FSHP + as.factor(AA)"



################################################################################
# Read all the experimental CSV files and apply the models.
################################################################################
exp_data = read_data_from_files(args[2])
count = c()

i = 1
for (model in models) {
    count[i] = apply_model(model, exp_data)

    if (i >= 12)
        save_predicted_values(i, model, exp_data)

    i = i + 1
}


################################################################################
# Print the accuracy
################################################################################
cat("start_accuracy_values\n")
total_count = length(exp_data$REL)
for (i in 1:length(model_names)) {
    cat(i)
    cat(" ")
    cat(model_names[i])
    cat(" ")
    cat(count[i] / total_count)
    cat("\n")
}

################################################################################
# Author: Reecha Nepal
# Date: August 13, 2012
# Purpose: This script runs all the other scripts (blast, density, etc..). It
# creates a CSV file for each protein.
```

```
# To run this script do the following:
#    python run_all.py pdb_names.txt
#    or
#    python run_all.py 119LA
# The output files will be created in a directory called "csv".
# File: run_all.py
################################################################################

import os
import sys

import A_Download_Blast.download_gi_number
import A_Download_Blast.download_fasta
import A_Download_Blast.download_blast
import B_Download_Density.download_mmCIF
import B_Download_Density.calculate_density
import C_NACCESS.download_pdb
import C_NACCESS.run_naccess
import C_NACCESS.extract_data
import D_Entropy.extract_data
import E_Misc.align_tables
from multiprocessing import Pool

################################################################################
# Gets the output path for the given pdb_name. For example, if the PDB name
# is 1HGXA then the output path would be:
#     csv/1HGXA.csv
################################################################################
def get_output_path(pdb_name):
    output_path = os.path.join("csv", pdb_name + ".csv")
    return output_path

################################################################################
# Get a list of pdb names from the a file.
################################################################################
def get_pdb_list():
    if len(sys.argv) != 2:
        print "To run this script do the following:\n" \
              "\tpython run_all.py pdb_names.txt\n" \
              "\tor python run_all.py 119LA"
        sys.exit(-1)

    pdb_list = []
    if os.path.exists(sys.argv[1]):
      input_file_handle = open(sys.argv[1], "r")
      lines = input_file_handle.readlines()
      for line in lines:
          pdb_name = line.strip()
          if len(pdb_name) > 1:
              pdb_list.append(pdb_name.upper())
    else:
      pdb_list.append(sys.argv[1])
    return pdb_list

################################################################################
# Gets the path to folder that contains the naccess code.
################################################################################
def get_python_code_path():
    script_path = os.path.realpath(__file__)
    parent_directory = os.path.dirname(script_path)
    return os.path.dirname(parent_directory)

################################################################################
# Runs the main script.
################################################################################
def save_data_to_csv(pdb_name, data, output_path):
    f = open(output_path, 'w')
```

```
    f.write('Num,denB,RES,E,E6,FSR,FA,FG,FSHP,EntropyQueryLetter,AA,REL,CATH\n')
    index = 0
    for record in data:
        index = index + 1
        f.write('%d,' % (index))
        f.write('%s,' % record['denB'])
        f.write('%s,' % pdb_name)
        f.write('%.3f,' % record['E'])
        f.write('%.3f,' % record['E6'])
        f.write('%f,' % record['FSR'])
        f.write('%f,' % record['FA'])
        f.write('%f,' % record['FG'])
        f.write('%f,' % record['FSHP'])
        f.write('%s,' % record['QueryLetter'])
        f.write('%s,' % record['AA'])
        f.write('%s,' % record['REL'])
        f.write('%s' % record['CATH'])
        f.write('\n')
    f.close()


################################################################################
# Runs the main script.
################################################################################
def run(pdb_name):
    output_path = get_output_path(pdb_name)
    if os.path.exists(output_path):
        return

    A_Download_Blast.download_gi_number.run(pdb_name)
    A_Download_Blast.download_fasta.run(pdb_name)
    A_Download_Blast.download_blast.run(pdb_name)
    B_Download_Density.download_mmCIF.run(pdb_name)
    B_Download_Density.calculate_density.run(pdb_name)
    C_NACCESS.download_pdb.run(pdb_name)
    C_NACCESS.run_naccess.run(pdb_name)

    (density_table, entropy_table) = \
        D_Entropy.extract_data.GetEntropyAndDensityValuesForPDB(pdb_name)
    naccess_table = C_NACCESS.extract_data.GetNaccessValuesForPDB(pdb_name)
    aligned_table = E_Misc.align_tables.align_tables(density_table,
                                                     entropy_table,
                                                     naccess_table)
    save_data_to_csv(pdb_name, aligned_table, output_path)

################################################################################
# The main function, this gets run first when the program is run from the
# command line.
################################################################################
if __name__ == "__main__":
    python_code_path = get_python_code_path()

    # put the file in a directory named "csv"
    if not os.path.exists("csv"):
        os.makedirs("csv")

    pdb_list = get_pdb_list()
    pool = Pool(processes=8)
    pool.map(run, pdb_list)
    #for pdb in pdb_list:
    #  print pdb
    #  run(pdb)

################################################################################
# Author: Reecha Nepal
# Date: February 5, 2012
# Purpose: This program takes entropy data for multiple residues and for each residue
```

```
# creates a filter matrix. For example, if residue 119LA had entropy data for
# 124 amino acids then a 124x124 matrix would be saved in matrix/119LA.csv.
#
# Required input file for this program:
#   Brel2Data215.csv
# To run this program do the following:
# > R -f tertiary_contact_filter2.R
# File: tertiary_contact_filter2.R
################################################################################

################################################################################
# Read the CSV file
################################################################################

# replace this with file.choose() to manually choose the CSV file
file_name = "Brel2Data215.csv"
csv_data = read.csv(file_name, sep= ",", header =TRUE)

# Save the matrix output to a folder named "matrix"
dir.create("matrix", showWarnings=FALSE)

################################################################################
# For each residue (119LA, 153LA, etc...) compute the filter matrix
################################################################################
residue_list = unique(csv_data$RES)
for (residue in residue_list) {
  residue_data = subset(csv_data, RES==residue)
  AA = residue_data$AA
  Entropy_20 = residue_data$E

  len = length(AA)
  m = matrix(nrow=len, ncol=len)
  for (i in 1:len) {
    current_entropy = Entropy_20[i]
    for (j in 1:len) {
      new_entropy = Entropy_20[j]
      multiplied_entropy = (current_entropy + new_entropy)/2
      if (i == j)
        m[i, j] = "N/A"
      else if (multiplied_entropy < 1.3025)
        m[i, j] = 0
      else
        m[i, j] = 1
    }
  }

  # Add amino acid names as heads (in the first row and first column)
  header_row = c()
  for (cur_aa in AA) {
    header_row = c(header_row, toString(cur_aa))
  }
  m = rbind(header_row, m)
  # Add one extra item to the column since the size of the matrix is 1 biger
  header_col = c("", header_row)
  m = cbind(header_col, m)

  # Save the matrix to a csv file (119LA.csv, etc...)
  file_name = paste(residue, ".csv", sep="")
  file_path = file.path("matrix", file_name)
  write(m, file_path, ncolumns=len + 1, sep=",")
}

################################################################################
# Author: Reecha Nepal
# Date: May 6, 2012
# Purpose: This program calculates the relative solvent accessibility (REL)
# using the following model:
```

```
#   REL ~ E20 + E6 + FSR + FSHP + as.factor(AA)
# The coefficient for the above model are caclulated using training data
# (Brel2Data268.csv). This model is then applied to the experimental data
# (Brel2Data215.csv) to predict new REL values.
#
# The predicted REL values and the actual REL values are then converted to
# binary using the following forumal:
#   if (binary_actual_REL >= 20)
#     binary_actual_REL = 1
#   else
#     binary_actual_REL = 0
#
#   if (binary_predicted_REL >= 20)
#     binary_predicted_REL = 1
#   else
#     binary_predicted_REL = 0
#
# This accuracy of binary_predicted_REL vs binary_actual_REL is then printed.
#
# Additonally this program can also apply a tertiary contact filter. See
# bellow for how the filter works.
#
# Required input files for this program:
#   Brel2Data215.csv
#   Brel2Data268.csv
#   matrix/ (output from tertiary_contact_filter2.R)
# To run this program do the following:
# > R -f accuracy.R
# File: accuracy_with_filter_2.R
################################################################################

################################################################################
# Read the training and experimental data.
################################################################################
# replace this with file.choose() to manually choose the CSV file
file_name_268 = "Brel2Data268.csv"
file_name_215 = "Brel2Data215.csv"
blast_268_data = read.csv(file_name_268, sep = ",", header = TRUE)
blast_215_data = read.csv(file_name_215, sep = ",", header = TRUE)
blast_268_data_frame = data.frame(E20 = blast_268_data$E,
                                  E6 = blast_268_data$E6,
                                  FSR = blast_268_data$FSR,
                                  FSHP = blast_268_data$FSHP,
                                  AA = blast_268_data$AA,
                                  REL = blast_268_data$REL)
blast_215_data_frame = data.frame(E20 = blast_215_data$E,
                                  E6 = blast_215_data$E6,
                                  FSR = blast_215_data$FSR,
                                  FSHP = blast_215_data$FSHP,
                                  AA = blast_215_data$AA,
                                  REL = blast_215_data$unREL,
                                  RES = blast_215_data$RES)

################################################################################
# Use our model to predict the REL values for the experimental data.
################################################################################
lm_268 = lm(REL ~ E20 + E6 + FSR + FSHP + as.factor(AA), blast_268_data_frame)
predicted_215_rel = predict(lm_268, blast_215_data_frame)

################################################################################
# Map REL values to binary.
################################################################################
binary_actual_REL_215 = c()
for (rel in blast_215_data_frame$REL) {
  if (rel >= 20)
    X2 = 1
  else
```

```
    X2 = 0
  binary_actual_REL_215 = c(binary_actual_REL_215, X2)
}

binary_predicted_REL_215 = c()
for (rel in predicted_215_rel) {
  if (rel > 23)
    X2 = 1
  else
    X2 = 0
  binary_predicted_REL_215 = c(binary_predicted_REL_215, X2)
}

#############################################################################
# Calculate accuracy
#############################################################################
match_count = 0
len = length(binary_actual_REL_215)
for (i in 1:len) {
  if (binary_actual_REL_215[i] == binary_predicted_REL_215[i]) {
    match_count = match_count + 1
  }
}
accuracy = (match_count / len) * 100
print(accuracy)

#############################################################################
# Apply the tertiary contact filter.
#############################################################################
filtered_values = binary_predicted_REL_215
residue_list = unique(blast_215_data_frame$RES)
for (residue in residue_list) {
  print(paste("Processing matrix: ", residue))
  residue_index = 0
  data_len = length(blast_215_data_frame$RES)
  for (i in 1:data_len) {
    if (blast_215_data_frame$RES[i] == residue) {
      residue_index = i;
      break;
    }
  }

  matrix_file_name = paste(residue, ".csv", sep="")
  matrix_file_path = file.path("matrix", matrix_file_name)
  filter_matrix = read.csv(matrix_file_path, sep=",", header=FALSE)

  row_count = length(filter_matrix)
  col_count = row_count

  for (row in 1:row_count) {
    for (col in 1:col_count) {
      distance = 0
      if (row > col) {
        distance = row - col
      } else {
        distance = col - row
      }

      if (filter_matrix[row, col] == 1 andand distance > 10) {
        filtered_values[residue_index + row] = 1
        filtered_values[residue_index + col] = 1
      }
    }
  }
}

#############################################################################
```

```
# Calculate accuracy with the filter applied
################################################################################
match_count = 0
for (i in 1:len) {
  if (binary_actual_REL_215[i] == filtered_values[i]) {
    match_count = match_count + 1
  }
}
accuracy = (match_count / len) * 100
print(accuracy)
```

B. 1363 PDB Table

Table B.1: List of 1363 PDB IDs used in the 1363 training data set.

2VB1A	1KQPA	1R0RI	2R2ZA	1OQVA	1V9YA	2GYQA	1O04A	1UPQA
2DSXA	1UOWA	1JM1A	2BMOA	1F4PA	1PLCA	3EJVA	2F22A	3SEBA
1R6JA	3C70A	1K7CA	1FYEA	1G61A	1KKOA	2P8IA	1PZ7A	3C9UA
2B97A	1PSRA	1H1NA	1NZ0A	2C4BA	1VH5A	1U7IA	2HQXA	1WPUA
1GCIA	1YQSA	1SU7A	1TU9A	1FCYA	1UTGA	2FTRA	2CJTA	1H41A
2F01A	3SILA	2EUTA	1QU9A	1XUBA	1EAJA	2J5YA	1E6UA	1UUQA
1G6XA	1FSGA	1U07A	1QW9A	1K3YA	2IYVA	1ODZA	1JBOA	1OF8A
1MUWA	1R2QA	1P6OA	2A26A	2PRVA	2FS6A	2J73A	2GDGA	1HD2A
1DY5A	1RQWA	2UUYA	2IFQA	1OX0A	1WDDA	1EZGA	1Z72A	1OCYA
1G66A	1LQTA	2V7FA	1I24A	1WPNA	1WDDS	1M1FA	1FM0D	1OFZA
1IX9A	1M2DA	1I8OA	1O7IA	1LQ9A	2PHNA	1Q5YA	1WL8A	3BI1A
1VYRA	1EUWA	1VZIA	1V8HA	1NXMA	1U9CA	2HIYA	1W1HA	1ZKPA
2BW4A	2AXWA	1I4UA	1LC0A	1J2RA	2FXUA	256BA	1KW3B	2O02A
1OEWA	1KJQA	1XODA	1MJUL	2F69A	1WZDA	2ACFA	1Q0RA	2OPLA
1N9BA	1PMHX	1H4AX	1W7CA	1FLMA	2PIEA	2IVYA	2O9CA	2FNUA
2PVBA	2C9VA	2D8DA	1OI7A	1R0MA	1P4CA	2HOXA	2HEWF	1LV7A
2PPNA	1XJUA	1M1NB	1WMAA	2I6CA	1PZ4A	2UX9A	2GLZA	1KM4A
2FDNA	1JBEA	1M1NA	1USCA	1SG4A	1UZKA	1L6RA	1ABAA	1VL7A
1NLSA	1UWCA	1HBNA	1M4LA	1KMTA	1PINA	1YE8A	1XD3A	1J8UA
2GUDA	1W23A	1HBNB	1RWHA	2NR7A	3BUXB	1K3IA	1XD3B	1P3CA
7A3HA	1DS1A	1HBNC	1GNLA	2Q9OA	1IJYA	1KS8A	1GVDA	2PBDP
1RTQA	1K5NA	1H4XA	3CHBD	2GF3A	1KUFA	1M55A	1OS6A	1M7JA
1NKIA	1QLWA	1X9IA	2HBAA	1RYAA	2AKZA	1UQ5A	1JI7A	1DJ0A
2Z6WA	1N62B	1C9OA	1K3XA	1JNDA	1RHSA	2GMNA	1DI6A	2HSJA
1AHOA	1N62C	1KQ6A	1EAQA	2NLVA	1VIMA	1Y8AA	2CVDA	1X82A
1XG0C	1N62A	1UWKA	1OOHA	3BBBA	1JO0A	2QFAA	1HZTA	2BT6A
1XG0A	1ZL0A	2CS7A	1MF7A	1O8XA	1F1EA	1EW4A	1W5RA	2J2JA
2NMZA	1ITXA	1WRIA	2C60A	3D32A	3CJSB	1UWWA	1THFD	2F62A
1IXHA	1I40A	1MJ4A	2B82A	1HYOA	3CJSA	1JUBA	1IDPA	1O9GA

1MWQA	2V9VA	1VR7A	2FVVA	1ES9A	1GVEA	1T7RA	1IO0A	2IMJA
1P1XA	2V89A	1WM3A	2QJZA	2FCJA	1RYQA	2C78A	1V4PA	1KQ3A
1JFBA	2BHUA	1XDNA	2TPSA	1I0DA	1JKVA	1PFBA	1URSA	1ISUA
1O7JA	1G8TA	1KG2A	1C1DA	1HDHA	2BFDA	2TNFA	2RL8A	2BRJA
1GA6A	1SBYA	2BLNA	2G3RA	1UCDA	2BFDB	1NYCA	1VE1A	2J8KA
1Q6ZA	1Z2UA	1UCRA	2GOMA	1V70A	1N8VA	2OHWA	1VMGA	2Z3GA
2RBKA	2ABSA	1Q6OA	1O8BA	2GECA	1I1JA	1PWBA	1HNJA	1XRKA
2JHFA	1LS1A	2VPAA	1QAUA	1VDWA	1SJYA	1GP0A	1QREA	1CRUA
2IIMA	1CTJA	1NWWA	1RDQE	2PV2A	1WB4A	1NG6A	1G3PA	2A35A
1NQJA	1BKRA	2IJ2A	1GXUA	1VD6A	2G84A	1ES5A	2R8OA	2PYQA
1OAIA	1T2DA	1W6SA	2J8WA	1JL1A	2ODKA	1FP2A	1JU2A	2IBAA
1C7KA	1UZ3A	1W6SB	1GKPA	1GXMA	2G7SA	2POFA	1O4YA	2FOMB
1CXQA	1RG8A	2DLBA	2AEBA	1E7LA	1K0MA	1XSZA	1VM9A	3ER7A
1JR8A	1F0LA	2F1KA	1HFES	1WQJB	2JE6I	2AALA	1VJOA	1PFVA
1IRQA	1SMBA	2F6UA	2EZ9A	1WQJI	1Y1PA	2D29A	2JE8A	1V58A
1CIPA	2ZKDA	2HX0A	1MXGA	1V2BA	1RYLA	1VKFA	2A9DA	1WO8A
1NTVA	1PTFA	2V33A	3ETJA	1G8QA	1JKXA	2QEEA	2GDMA	1B2PA
1SX5A	1Y7BA	2NMLA	2GUIA	1B5EA	1I58A	1HX6A	2OD4A	3BYPA
1JX6A	2FHPA	1MG7A	1K92A	1IU8A	1N08A	2EX0A	2NXFA	1YDYA
1JYKA	2D5JA	1IX2A	2PY5A	1LQVA	1XPPA	1Y7TA	1EEXA	1XSVA
1FL0A	1RKIA	1JKEA	1U8VA	1J1NA	1KAFA	1OK7A	1EEXB	1RXQA
1Y5HA	2FL4A	2HHVA	1S1DA	1QGIA	1YD9A	2EV0A	1EEXG	1BUPA
1GMUA	1W0PA	1A62A	2VLQB	1WS8A	1M4JA	2DEKA	2ONFA	1NVMB
2INWA	2G5RA	1UGIA	2VLQA	1YXYA	1U60A	1UEBA	2RLDA	1DC1A
1V2XA	3DMCA	1OQJA	1D3GA	1T9IA	1XKGA	1VHQA	2ID3A	1XCRA
1QS1A	2FHFA	1BI5A	1TJOA	3C8WA	2QFKA	1GXRA	1VCLA	1OGQA
1NLQA	2FMPA	2V3ZA	1P0ZA	1QMGA	2ZKMX	1VI0A	1JHDA	2FA1A
1OMRA	1FDRA	1VKEA	2ITEA	1YT3A	2CJ4A	1UFIA	2R85A	1QD1A
1C4OA	1OQ1A	1SXRA	2BZ6L	1E2WA	1E5PA	1FXOA	1YOCA	1H03P
3BEXA	1DMHA	2AFWA	1F74A	2IU5A	2ICTA	1VPMA	1U5DA	1G3MA
2OOCA	1S5UA	2BKFA	1T92A	1PE9A	2FBNA	1GZGA	1WDVA	1HZ6A
1C4QA	1T1JA	2F9HA	1T4BA	2BDRA	1PU6A	1L3LA	1P3DA	1RH6A
2FSRA	1NKRA	1JOVA	1X7DA	1J2JA	1G8KA	2AJ7A	1D1QA	1DUVG
2JFGA	1NSZA	2GUYA	1S95A	1WLZA	3ELGA	1N1JA	2IDLA	2DUCA
1QQ5A	2O62A	2IA1A	2NW8A	1ZKEA	1YLLA	2R7GA	2BKYA	2NUGA
1W7BA	2I6HA	1VIOA	1FIUA	1J2JB	1VPDA	1UD9A	1GU7A	1T82A
2R1JL	2J9UA	1K7IA	1TQJA	1GY7A	1DWKA	1TH7A	2AVDA	1IGQA
1L2HA	2J9UB	2NNUA	1Y0UA	1RFYA	2G50A	1JP4A	1P1JA	3BOFA
1VHWA	1GXJA	1CCWB	1T0AA	1T1VA	1W2YA	2I8DA	1OOYA	1UV7A

2B5AA	1WMXA	1CCWA	1X6OA	2CPGA	1IM5A	1JY1A	2EG6A	1SGWA
2GHSA	1G73A	2OKFA	1OW4A	1O26A	2G82A	2HFTA	1JW9B	1VFJA
1SFXA	1JC4A	1LQAA	1KQFA	1M7YA	1O54A	2QIYC	2IGIA	1DQGA
2F71A	1DQTA	1T9HA	2P0SA	2F23A	2P97A	3BYQA	1QXMA	1P99A
2BEMA	1X0PA	2APJA	1JNRA	1WC9A	3CLSD	1T0BA	2DVTA	1V9FA
1WBHA	2PI2A	2I0KA	1ELWA	2V5ZA	3CLSC	1YKIA	1D02A	1KZQA
1W6GA	1TJLA	2BO9B	1L9XA	1JHJA	1QSAA	2I5IA	2VAPA	1XKPA
1MV8A	2PI2E	2CL5A	2J43A	2OY9A	2IMLA	2CCHB	1MTYD	1P5VA
2FP1A	1H3FA	1VKIA	1UKKA	1OU8A	1KOLA	1TXGA	1MTYB	2OX6A
1EJDA	1YOZA	2OMZA	1AOPA	1IT2A	1NNHA	2B8MA	2CB2A	1ZMTA
2ELCA	1F2LA	2OMZB	1V7ZA	2B9DA	1S99A	1Q8FA	1MTYG	1K94A
1VLPA	2OX7A	2NP5A	2FBHA	2A1HA	1Q33A	1T15A	2JDIA	1CB8A
2I74A	2GIYA	1TDZA	1OB8A	1A73A	2QJ2A	3ENBA	2JDID	1UHGA
2F1FA	2BS2A	1Y0BA	1Q9UA	1I4JA	1GQEA	1U7KA	1Q0QA	1T06A
1KPTA	2BS2C	1BEBA	3ERJA	1PVGA	2JI7A	2B3YA	1CHMA	2PF5A
1JAKA	2BS2B	2IT9A	1ORUA	1V4VA	1B65A	1CQ3A	2JDIG	1FL2A
2FPWA	1X74A	1N7HA	2VO9A	1Q0PA	1RY9A	1VCTA	2JDIH	1M6YA
1CV8A	2NT0A	1CMCA	2O70A	1KXOA	1WURA	2H62A	1QB5D	1YQGA
2O3FA	2PBLA	1FNLA	1QWRA	2B8UA	1H8PA	2CWZA	2JDII	1AYOA
1PG4A	1O5KA	1WEHA	2PNXA	1UUZA	1VLRA	1J75A	1UYPA	1OBBA
1PM4A	1VK8A	2C2IA	2GDQA	1K6KA	1H7EA	2QMMA	2ZQNA	1LR5A
1ZARA	2NACA	1VC4A	1M6SA	1IQ4A	1O5UA	1EYQA	1UNNC	2PEQA
1K75A	1VMEA	1JIDA	1F1MA	1FPOA	1TX2A	2CXKA	1REGX	1G3KA
1UUJA	2AC0A	2OOKA	1V54A	1IG0A	3E5UA	2PD1A	1VQ3A	2FA8A
1NBCA	1M3UA	1IIBA	1UXYA	1FS1B	1O69A	1ZPDA	1YARO	1OTFA
1GTFA	1NARA	1C3DA	1V54C	1G8EA	2DYJA	1JB7A	1YARA	1VBKA
1NE7A	1XFFA	1SW5A	1V54B	1KU3A	1JR2A	1TZZA	2I9AA	1J9JA
1YRBA	2CX1A	1NZYA	1VCAA	1FS1A	1SS4A	2FZPA	1XEDA	1AGQA
1TVXA	1LBUA	1VYBA	1V54D	1NKPA	1SGJA	2A2MA	1Q08A	1BXYA
2EX4A	2RGQA	2FURA	1V54E	2FMMA	1VKHA	2PQRA	2FREA	3CPTA
2NPTA	1TR0A	2AG4A	1V54F	2ZD1B	2OTMA	2PQRC	3ECFA	3CPTB
2FUEA	1ZLQA	1BDOA	1V54G	2NS9A	1WVEA	2IGTA	2NR5A	2NYCA
1DQNA	1LSTA	1A3AA	1V54H	2CYYA	2OU3A	2GIAA	1YT8A	1U6ZA
1VH4A	2Q03A	2I9XA	1V54I	2R7DA	1B25A	1V3EA	1LUAA	1F0XA
1WWZA	1FN9A	1A9XA	1V54J	1RWZA	2VZSA	1M2TB	1O91A	1STMA
1NRZA	1YX1A	1OWLA	1V54K	1WZ3A	2CWYA	1VKCA	2DSYA	1M0DA
1GO3F	1RQPA	1A9XB	1V54L	2Z1CA	2CHOA	1JFRA	2A6CA	1CI4A
1XE7A	2H1TA	1DUSA	1V54M	1WZ8A	2CB5A	1OAOC	1U94A	3BDUA
1JLVA	1MGTA	1JH6A	1WDYA	1L8BA	1K8WA	1OJXA	2FEFA	1G291

1XKRA	1YTLA	1YKSA	1LM5A	1CS6A	1TUHA	1YQ2A	1QMYA	2GAUA
1L1LA	1VLAA	1H4RA	1UDZA	1Y9IA	2IEAA	1IOWA	1U7PA	1Y7PA
2P3PA	1GUQA	1WLGA	1LB6A	1V2ZA	1EX2A	2MNRA	1TZYC	1AMUA
3BWUF	1ATZA	1RWIA	2SPCA	1RKXA	2H7XA	1J0HA	1TZYA	1B63A
3BWUD	2O9AA	1WOQA	1NU4A	1SQDA	1LWDA	1ZQ9A	1TZYB	1XIWA
2GMQA	2HQYA	1UJ2A	1W2FA	1LUZA	2OD6A	1SZOA	1TZYD	1F00I
1YNBA	1MR7A	1XTTA	1K0IA	1H8UA	1V82A	2CWQA	2ODFA	1IWMA
1U9KA	2I02A	1QNAA	1N97A	1X8DA	1SSQA	2CH5A	1S4KA	2FF4A
1R7AA	3BB9A	1FXLA	2H26A	1VQQA	2AUWA	1V0EA	1ZPVA	2ZGYA
3BSWA	1C3CA	1MPGA	3PROA	1VHSA	1EXTA	1ON3A	1A8LA	1NNWA
1U55A	1RYIA	1W96A	3PROC	2GAXA	2Z3QA	1PL3A	2NZWA	1JYSA
1X7VA	1VKNA	1TENA	2G5FA	2GU2A	2Z3QB	2OKUA	1CEOA	1DZFA
1Z6OA	1TXOA	1W85I	1HRUA	1WIWA	1VCHA	2BPTA	1DEKA	1MZGA
1Z6OM	1KHYA	1PPRM	1I36A	1ROWA	2OGQA	2SQCA	1N2ZA	1U5UA
2GF6A	1JGTA	1SACA	1UANA	1TYZA	1XHNA	2IHTA	1WDJA	1EL6A
2RDEA	1YNHA	1Z2WA	1W07A	1Q2YA	2BO4A	2NUJA	1KXGA	1SD4A
1ZELA	1T6SA	1RFXA	3C10A	1UC8A	1I7QA	2EBNA	1ITUA	1KNQA
1WC1A	2APOB	2SCPA	2IB0A	2CU6A	2F2HA	1YRRA	1XV2A	1H2VC
1EKJA	1M1HA	1SR4A	2PULA	2ES9A	1PN2A	2PV7A	1AOCA	1D2OA
1KWAA	2I5NM	2FNOA	1VC1A	1TE5A	1G5HA	2FEAA	1S1QA	1OEYJ
1DAAA	1M5WA	2PIAA	1TKIA	1K32A	1PUCA	3CNVA	1RLJA	1OEYA
1JZTA	2D7VA	1NBAA	1CSNA	1SGMA	2P5MA	3CW9A	1OFDA	1G3QA
1T6NA	1JBSA	3DCXA	1MBMA	1L2WA	2AEEA	1DQAA	1SZWA	1EKEA
1SVMA	1VQZA	1EX0A	1XIZA	1L2WI	1ITWA	2OAFA	1OTKA	2HKUA
2PSPA	2HLJA	1V6ZA	1Q4GA	1S12A	1JRLA	1FP3A	2OWPA	1XVHA
1Q7FA	1VLFM	2D0OA	1GL4A	1RIFA	1J5WA	1SR9A	1COZA	1D3BB
1GVNA	1VLFN	1DP4A	1B9HA	1R7LA	1OGDA	2HKJA	1D3YA	2FKZA
2O34A	2IFXA	1HBKA	1YVIA	2FNAA	1OR7C	1FC3A	1E8CA	1U5KA
1SA3A	1F61A	1P42A	1HE1A	1R0VA	1W94A	2LIGA	2BM5A	1OR7A
2FIUA	2HEKA	2CZVA	1U9LA	2I7NA	1I0RA	1J3WA	3E9VA	
2ARCA	1NN5A	2CZVC	1D2TA	1RYP2	1D7PM	2OFYA	1GTTA	1F5NA
2CYJA	2HQSA	1S14A	1F0KA	1RYP1	1PX5A	1ZPSA	1VP6A	1F3UB
1V5IB	2HQSC	2I9FA	1E3OC	1RYPL	1Z4RA	1MVEA	1R1TA	1F3UA
1BTEA	1PVMA	1G7SA	2FBLA	1RYPJ	1NQUA	1OA8A	2HAZA	1VR9A
1GUTA	2O6PA	1UQTA	1I7NA	1RYPK	3BLZA	2AEUA	1V71A	1SG6A
1M6KA	2PA7A	1ZC3B	1OISA	1Q2HA	1T0TV	2B5GA	1MK4A	2BBKH
2ZDPA	1V5VA	1O75A	1SEIA	2BF5A	3B5EA	1JX4A	1UPTB	2BBKL
1FJ2A	1SH8A	1UW4B	1ETXA	1GUDA	1JYOA	1EZ3A	2DSTA	1ELKA
2O1QA	1EJ0A	1TU1A	1J5PA	1GXYA	1JYOE	3CX5I	2GA1A	1V6SA

1MLAA	1F46A	1UW4A	1R0DA	1D0QA	1YG6A	3CX5C	3CX5B	1G2QA
2BAYA	1A4IA	1RQJA	1K3SA	1JMKC	1C8UA	1R5LA	3CX5D	1PUFB
2BMWA	1INLA	1VI6A	2V94A	2PBKA	2GSVA	2ZGWA	3CX5F	1N2AA
2I53A	1H32A	1RKTA	1SQWA	1ZRUA	1C7NA	2H6FB	3CX5G	1EERB
1L7AA	1CS1A	2NZCA	1Y60A	2F1NA	1Q8RA	2H6FA	3CX5H	3CX5A
2ASBA	1DQZA	2QWXA	1DFMA					