Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

Protein Intrinsic Disorder Detection Based on Structural Features

Master Candidate

Alberto Crivellari

Student ID 2061934

Supervisor

Prof. Alexander Miguel Monzon

University of Padova

Co-supervisor

Prof. Damiano Piovesan

University of Padova

Academic Year
2022/2023

To my girlfriend,
my parents and friends.

Abstract

Structural Bioinformatics is a branch of science that involves the analysis of three-dimensional structures of molecules using computer science techniques. Initially, the primary focus was on proteins with a fixed three-dimensional structure. However, researchers in the last 20 years have shifted their attention to Intrinsically Disordered Proteins (IDPs), which are proteins containing disordered regions that exhibit highly heterogeneous conformations. The work in this thesis is centered around recognizing IDPs from sequences through the extraction of features that indicate protein disorder.

This work presents a software tool, AlphaFold-disorder (SASA), developed by implementing PSEA and SASA algorithms. Subsequently, the quality of results produced by the new software tool was compared with state-of-the-art software. The development process involved three major procedures: the implementation of the PSEA procedure for predicting secondary structures based on three-dimensional coordinates of amino acids; the implementation of the SASA procedure for computing RSA (Relative Solvent Accessibility) of amino acids using the SASA library; and the implementation of the FoldComp procedure for managing .fcz files, which are compressed protein files. To assess the quality of the results, the dataset was initially plotted to gain insights into the distribution of features. Machine learning models were then implemented. Finally, ROC and Precision-Recall curves between AlphaFold-disorder, AlphaFold-disorder (SASA), and the best machine learning model were compared. The comparison revealed that AlphaFold-disorder (SASA) predictions are on par with AlphaFold-disorder ones, while the machine learning model requires more training data to surpass their predictions.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Snippets

List of Acronyms

NMR Nuclear Magnetic Resonance

DNA DeoxyriboNucleic Acid

RNA RiboNucleic Acid

ATP Adenosine TriPhospate

IDP Intrinsically Disordered Proteins

DSSP Define Secondary Structure Prediction

SASA Solvent Accessible Surface Areas

CLI Command Line Interface

CSV Comma Separated Values

TSV Tab Separated Values

JSON JavaScript Object Notation

PIP Package Installer for Python

CAID Critically Assessment of protein Intrinsic Disorder prediction

PDB Protein Data Bank

mmCIF MacroMolecular Crystallographic Information File

SS Secondary Structure

SSE Secondary Structure Elements

RSA Relative Solvent Accessibility

pLDDT predicted Local Distance Difference Test

P-SEA Protein Secondary Element Assignment

ML Machine Learning

LASSO Least Absolute Shrinkage and Selection Operator

SVM Support Vector Machines

ROC Receiver Operating Characteristic

1 Introduction

1.1 BioComputing UP

BioComputing UP is a research laboratory and is part of the Department of Biomedical Sciences of the University of Padua. The research there is focused on developing bioinformatic tools and high quality computational methods which are put to use to solve important biological issues. Its main fields are: structural biology, functional biology, genome wide analysis, genetic diseases and cancer studies.

1.2 Internship description

The internship project I carried on during the stage at BioComputing Up was mainly about expanding a pre-existent software tool, Alphafold-disorder, with newer libraries. In particular, my tasks were:

- Studying Alphafold-disorder code and related papers, to understand the scientific principles beneath;
- Implementing an algorithm which detects protein secondary structure, based on atomic distances and angles, calculated with protein atomic coordinates and the library BioPython. This algorithm was described in a paper [11];
- Integrating this secondary structure detection algorithm into the software tool Alphafold-disorder;

- Developing a procedure to compute the solvent-accessibility for each aminoacid in the protein, using the algorithm devised by Shrake and Rupley in 1973, by integrating a specific library;
- Integrating this procedure to compute solvent-accessibility into Alphafold-disorder;
- Integrating the library FoldComp and developing a quick procedure to allow input files of the type .fcz in Alphafold-disorder. FoldComp is a library that creates .fcz files as a compression of normal protein data files, .pdb files. FoldComp library is described in a scientific paper [10];
- Evaluating results through plots and machine learning models.

1.3 Thesis Outline

This thesis is divided into chapters based on the 6 tasks I've accomplished in my internship.

Chapter 1 : brief introduction and contextualization.

Chapter 2: overview on proteins, starting from the basics concepts of biology, with a focus on proteins with intrinsically disordered regions, to how computer science is applied in this field.

Chapter 3: description of Alphafold-disorder and the relevant libraries involved, such as DSSP.

Chapter 4: development of the software tool AlphaFold-disorder (SASA). Focus on the three major procedures implemented.

Chapter 5: analysis of the results produced by the new software tool Alphafold-disorder (SASA) through plots and machine learning models, to assessment of the results produced.

Chapter 6: conclusions based on the insights provided by the plots made in the Chapter 5.

2

An Overview on Proteins

2.1 Basics of Molecular Biology

2.1.1 Cells

Life is made of cells, the fundamental working units of every living system. They are composed of water; macromolecules (proteins and polysaccharides) and other small molecules, such as lipids and amino acids.

Cells are the smallest structural unit of an organism capable of independent functioning.Each cell follows the same common cycle of birth, replication, protein synthesis and death.

2.1.2 DNA

The nucleus of the cell contains the DNA, which contains the genetic information of the individual, the DNA is highly folded to save up as much space in the nucleus.

The genome is an organism's complete set of DNA, the human genome contains about 3 billion DNA base pairs and 24 distinct chromosomes.

In figure 2.1 we can see the set of 24 chromosomes in the human genome. A cell normally contains 23 chromosomes, the 22 common ones, the autosomes, plus one of the sex chromosomes, "XX" in females and "XY" in males.

Chromosomes are contains pieces of DNA, among them we have genes: the basic functional units of heredity.

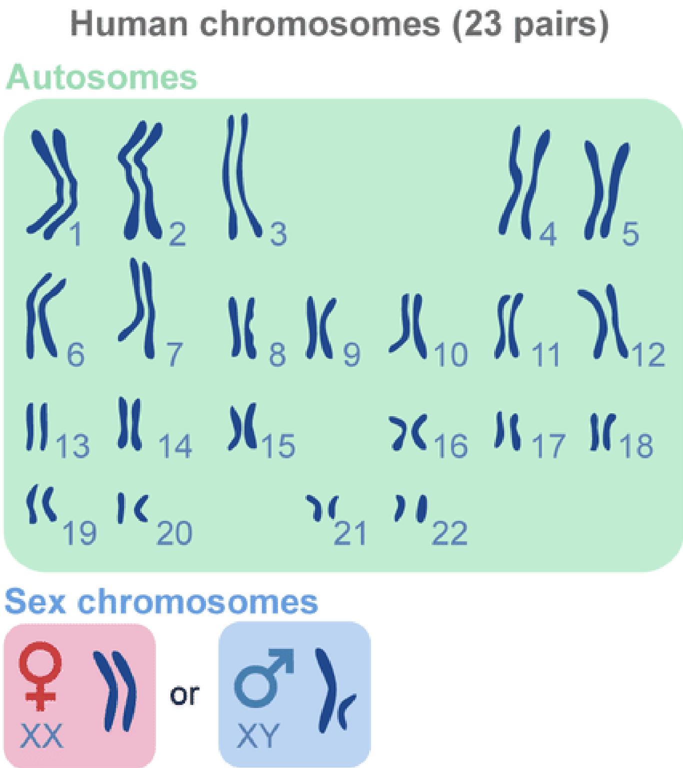


Figure 2.1: Chromosomes set in human genome, from [2]

Genes are specific sequences of DNA that encode instructions on how to make proteins, they determine the principal hereditary characteristics in the human being, such as height, muscular mass and appetite.

Mutations of one or more genes can cause alterations as harmless as a different eye color or as serious as a disease. These mutations can also provide beneficial effects, such as immunity or protection for some diseases, for example a specific gene mutation is known for its protection against malaria.

The DNA can be used to synthesize proteins, particular molecules that participate in most of the essential processes of the human body:

- building and repairing body structures;
- digesting nutrients;
- hormones: some hormones are proteins or protein-derived. Hormones are chemical messengers that flow through blood to coordinate different body's functions;
- executing various metabolic functions, or assisting the execution.

2.1.3 Central Dogma of Biology

The Central Dogma of Biology describes the transfer of genetic information from DNA to RNA to proteins. The key steps involved are :

- **Replication**: Duplication of DNA molecules, used during cell division;
- **Transcription**: Synthesis of an RNA molecule, using DNA as template;
- **Translation**: Synthesis of a protein, using the information encoded in RNA molecules.

Now I will describe the Transcription step, that produces RNA molecules and the Translation step, that synthesize proteins.

Transcription

Transcription is the process of copying a segment of DNA into RNA.

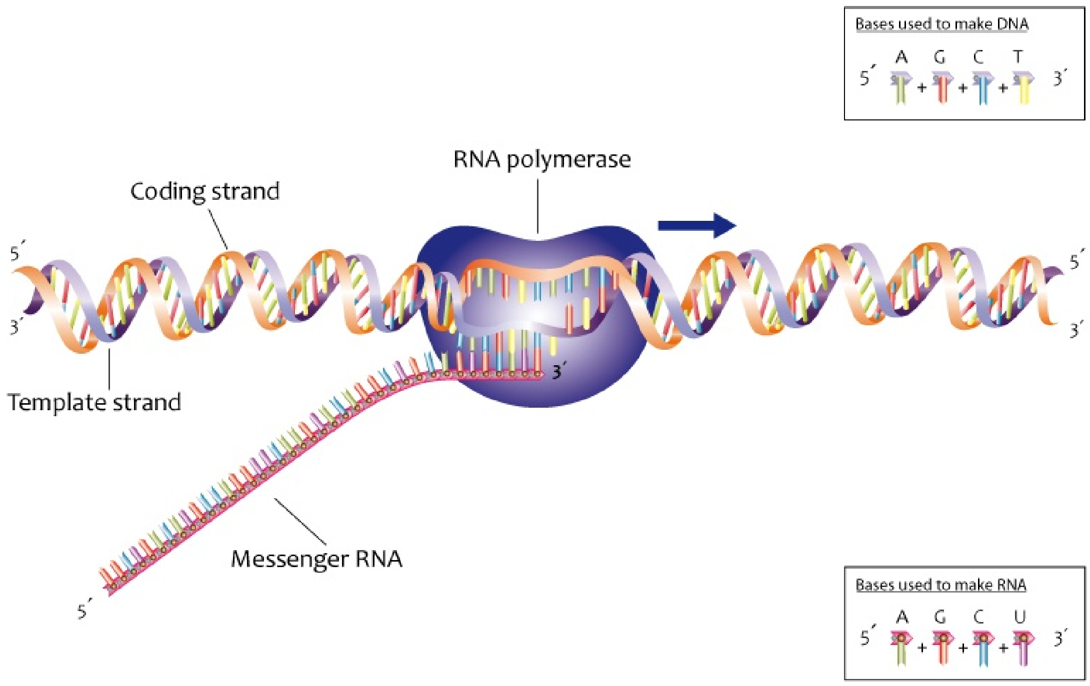


Figure 2.2: Transcription: RNA Synthesization, from [18]

An enzyme called RNA polymerase produces an RNA strand, by passing through a DNA strand.

The resulting RNA is the copy of the coding strand of the "input" DNA , while the strand used as template to create this copy is called template strand.

In figure 2.2 we can see how RNA polymerase creates the RNA. We can note that the RNA molecule is called messenger RNA, also known as **mRNA**.

There are other two types of RNA, tRNA and rRNA:

- **mRNA**: messenger RNA, contains the genetic information from the DNA. mRNA specifies ;
- **tRNA**: transfer RNA, transfer the correct amino acid to the ribosome. It acts as a bridge between the mRNA and the ribosome;
- **rRNA**: ribosomal RNA, combined with ribosomal proteins creates the ribosome.

Translation

Translation is the cellular process that synthesizes proteins based on the genetic instructions encoded in messenger RNA.

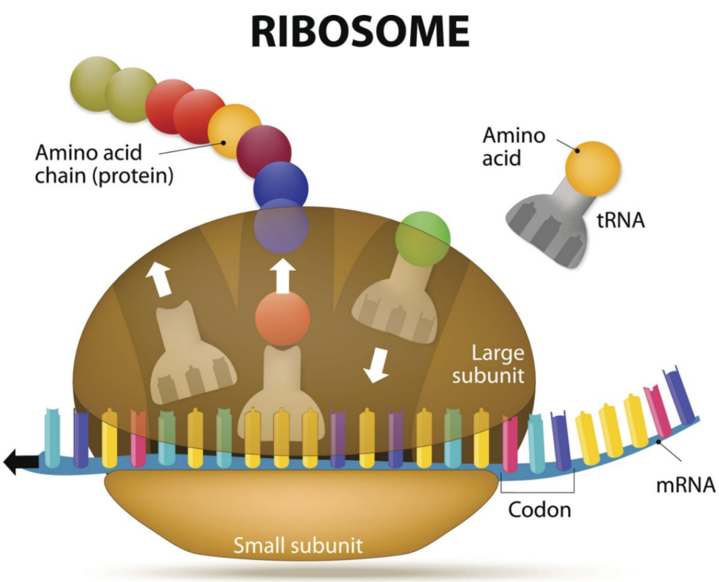


Figure 2.3: Translation: Protein Synthesization, from [5]

In figure 2.3 we can how the ribosome synthesizes the protein chain.

The synthesization happens inside a ribosome, which reads the triplet of nucleotides of the mRNA (**codon**) and brings the correct tRNA, which has a triplet of nucleotides (**anticodon**) that matches the codon.

Then the tRNA transfers its amino acid to the growing protein chain of the ribosome.

In figure 2.3 we can see how the ribosome, along with tRNAs, identifies the correct amino acid and transfers it into the growing protein, through a chemical bond.

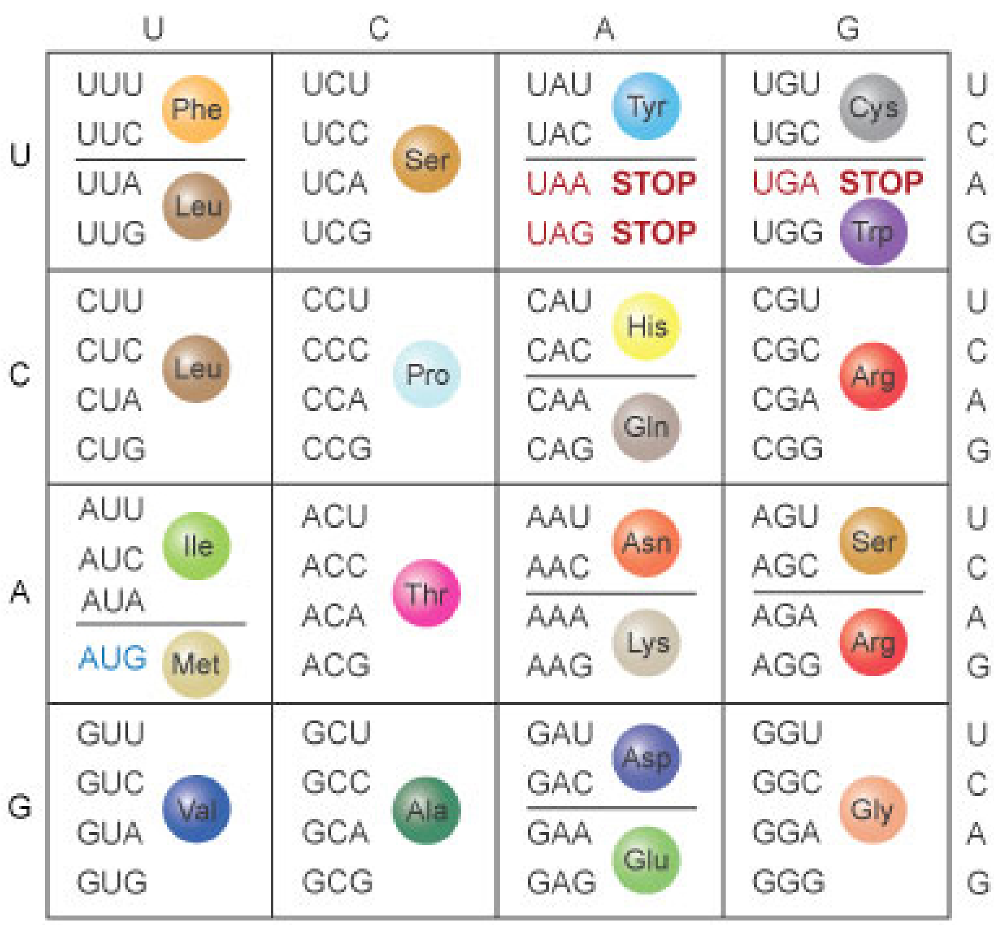


Figure 2.4: Genetic code, from [4]

In the figure 2.4 above, we can observe the Genetic Code: the set of triplets of nucleotides, codons and the corresponding amino acid.

We can observe 4 particular combinations of nucleotides that correspond to signal of start and stop the protein synthesization:

- The codon **AUG** identifies the **START** signal for the protein translation;
- The codons **UAA, UAG, UGA** identify the **STOP** signal, which ends the translation process producing the final protein.

2.1.4 Proteins

Proteins are large, complex molecules made up of amino acids, smaller subunits also referred to as residues. The residues are the building blocks of a macromolecule, so in this case the residues are the incorporated amino acids. Proteins are linear chains of different combinations of 20 different amino acids.

Protein Functions

- Cellular structure;
- Present in body's major components, such as skin and hairs;
- Hormones, some of them are proteins, they communicate with other cells;
- Enzymes are proteins, they regulate gene activity.

The protein function depends both on the amino acids' sequence order and types and on the three-dimensional structure the protein folds into.

Protein Folding

Proteins tend to fold into lowest energy three-dimensional conformation. They already begin to fold already when the amino acid chain is being formed during translation.

Different amino acids have different chemical properties and by interacting with each other the protein starts to fold adopting its functional structure.

The structure of the protein determines the protein function. Through folding some amino acids are more exposed, this determines which substrates the protein can react to.

Substrates are molecules or compounds that participate in a chemical reaction, they are starting materials or reactants which are acted upon by enzymes or catalysts.

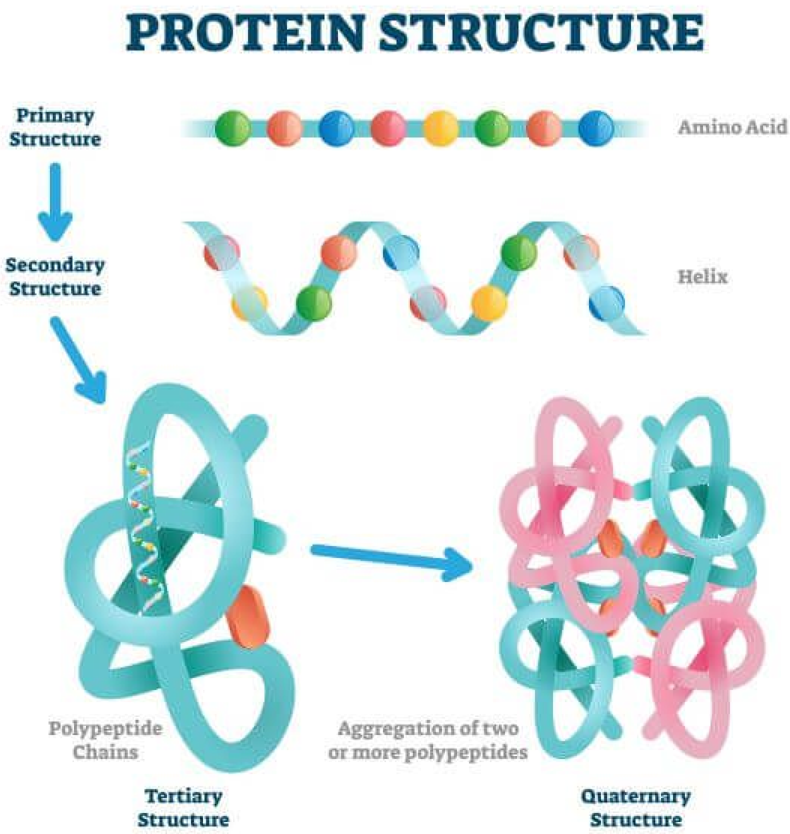


Figure 2.5: Structure of proteins, from [1]

From figure 2.5 we can see the four types of structures in a protein:

- **Primary structure**: the sequence of amino acids;
- **Secondary structure**: local structural patterns formed by residues;
- **Tertiary structure**: the global structure of the peptide chain;
- **Quaternary structure**: aggregation between the various peptide chains in the protein.

The main two types of secondary structures are :

- **Alpha-helix**: proteins bury most of their hydrophobic residues in the interior core, forming a spiral structure resembling an helical spring;
- **Beta-sheets**: segments of the protein are stretched out and aligned in a sheet-like arrangement.

Amino Acids

Amino acids are monomers[1] of proteins, each amino acid has a specific chemical behavior.

All amino acids in the human genetic code have a carboxyl group (-COOH) and an amino group (-NHH) bound to the central carbon atom. Amino acids differ for the side chain, while the carboxyl group and the amino group are the same for each one of the 20 amino acid. Side chains differ in these 3 features :

- three-dimensional structure;
- electric charge;
- hydrophobicity.

Amino acids are mainly classified by hydrophobicity :

- hydrophobic amino acids repel water, they are also called non-polar amino acids.
- hydrophilic amino acids are attracted to water, they are also called polar amino acids.

[1]A monomer in molecular biology is a molecule that can bond with other monomers to create a macromolecule.

Other classifications take into account the structure, functionality or electrical charge of the amino acid (uncharde, positively charged or negatively charged). Some other classifications are based on the particularity of the side chains, such as Sulfur-Containing.

Full Name	**Abbreviation (3 Letters)**	**Abbreviation (1 Letter)**	**Polarity**
Glycine	Gly	G	Non-Polar
Alanine	Ala	A	
Valine	Val	V	
Leucine	Leu	L	
Isoleucine	Ile	I	
Methionine	Met	M	
Phenylalanine	Phe	F	
Tryptophan	Trp	W	
Proline	Pro	P	
Serine	Ser	S	Polar
Threonine	Thr	T	
Cysteine	Cys	C	
Tyrosine	Tyr	Y	
Asparagine	Asn	N	
Glutamine	Gln	Q	
Aspartic Acid	Asp	D	
Glutamic Acid	Glu	E	
Lysine	Lys	K	
Arginine	Arg	R	
Histidine	His	H	

Table 2.1: List of Amino Acids

In the table 2.1, we can see the list of the 20 amino acids in the Genetic Code with :

- Polarity;
- Three letters abbreviation;
- One letter abbreviation.

To check which triplet of nucleotides corresponds to each amino acid in the Genetic Code, we can go back to figure 2.4.

2.1.5 Tandem Repeat Proteins

Tandem repeat proteins are the product of minimal folds from the repetition of simpler units. Their buried residues are more conserved, with a large surface and a high sequence variability.

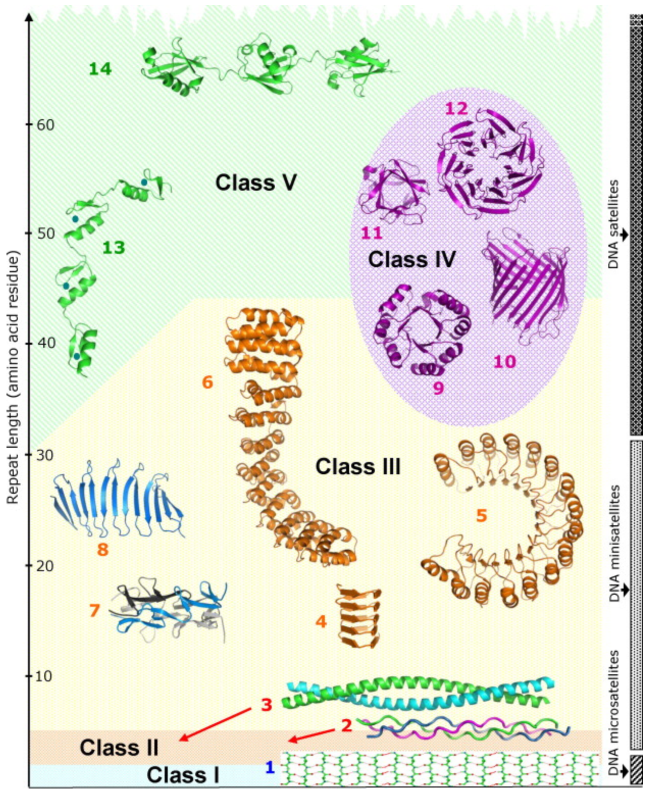


Figure 2.6: Tandem Repeat Proteins, from [9]

From the figure 2.6 we can see what tandem repeat proteins look like. We can also observe how they are divided into classes; tandem repeat proteins are classified according to periodicity in 5 classes:

- Class I: Aggregates;
- Class II: Collagen and Coiled-coils;
- Class III: Solenoids;
- Class IV: Toroids;
- Class V: Beads on a string.

2.1.6 Intrinsically Disordered Proteins

A polypeptide chain can be classified as different types of proteins, such as:

- Membrane proteins;
- Globular proteins;
- Tandem repeat proteins;
- Intrinsically disordered proteins.

The main difference between Intrinsically Disordered Proteins and the others is that IDPs don't have a fixed structure, either in same parts or in every part of the protein.

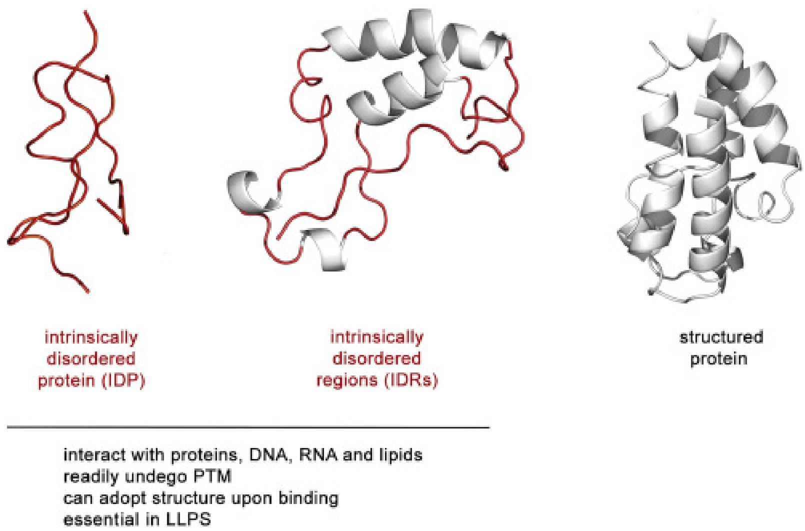


Figure 2.7: Intrinsically Disordered Proteins, from [13]

In disordered proteins the cost of folding is higher and so the protein has a lesser degree of freedom in folding.

It has complex surfaces which means high specificity with low versatility. For specificity we mean that when 2 proteins fit together, we need 2 highly specific protein structures to fit together, since their shape is so complex, an average protein can't be a proper fit.

Low versatility is the opposite: since the shape is quite complex, it doesn't fit together with most proteins, hence it's not versatile.

Due to their complex structures, disordered proteins are less prone to environmental stress: they can preserve their function in unstable conditions, such as high temperature.

In cell regulation around 25% of proteins are disordered, those are involved in highly dynamic and complex processes that require proteins with high specificity.

IDPs (Intrinsically Disordered Proteins) are really interesting to study, because they are implicated in several pathologies and due to their different functions they are involved in:

- Regulatory functions;
- Central role in the assembly of macromolecular machines, such as ribosomes;
- Transport of molecules through nuclear pore;
- Binding: IDPs can participate in one-to-many and many-to-many interactions, where one IDP region binds to multiple molecules, potentially gaining very different structures in the bound state.

They are also implicated in several pathological conditions, like cancer, cardiovascular diseases, and neurodegenerative diseases.

Some of the first classes of IDPs were notable due to their pathological roles in neurodegenerative diseases.

2.2 Computational Biology: Computer Science Applied to Biology

The growing amount of data in the field of Molecular Biology brought the scientists to exploit Computer Science. Computers can process data way faster than human beings, and especially in this field, where we have so much information on DNA, genes and proteins, it's really useful to compute and process these pieces of informations faster and possibly with less errors.

This new field takes the name of Bioinformatics, the union of Biology, Computer Science and Statistics. Statistics is needed because most of the information isn't 100% reliable, hence statistics methods are used. The main fields of bioinformatics are:

- **Structural data analysis**: its purpose is to predict the three-dimensional structures of proteins, nucleic acids and other biological molecules. Understanding the structure of these molecules can provide insights into their functions and interactions;

- **Omics data analysis**: for example genomics, proteomics, metagenomics, epigenomics and so on. Omics data is a broad term that refers to large-scale datasets generated from various biological technologies. These large-scale datasets enable researchers to study different aspects of biological systems, often on a global or comprehensive scale. For example, in genomics analysis, we have large-scale datasets focused on genome data, so we can analyze DNA sequences and genes organization and functions.

In this thesis the focus will be on structural data analysis, since the software tool I talk about analyzes the three-dimensional structure to obtain information on the disorder or binding propensity.

2.2.1 Structural Data

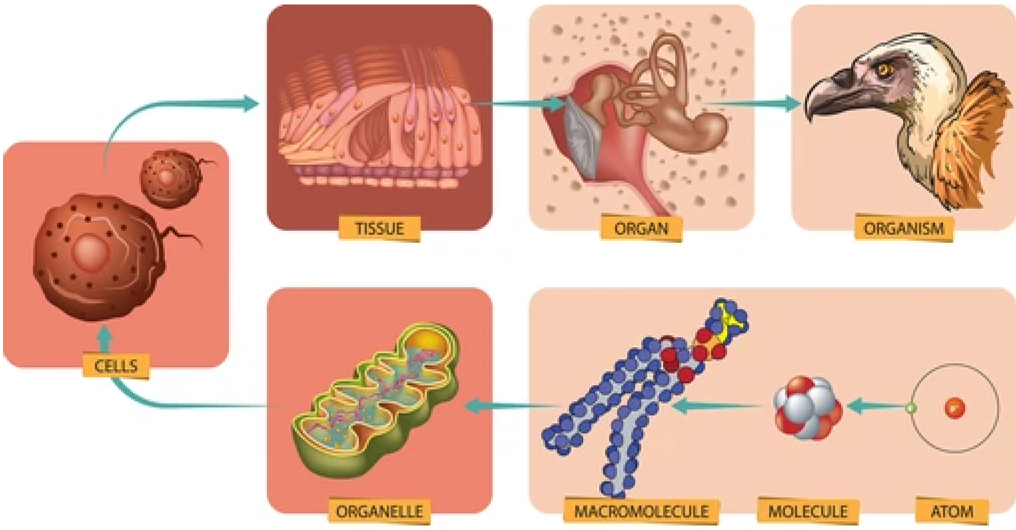


Figure 2.8: Atom to macromolecules to organism, from [12]

In figure 2.8 we can see the various steps from molecules to macromolecules to bigger complexes up to the final organism.

We can use structural data for:

- Structure prediction;
- Predicting protein interactions: protein folding, binding, protein assemblies;
- Structure comparison: by comparing the moolecules' structure we can determine to which species it belongs to and where in the evolutionary scale it stands;
- Exploring mechanisms of interaction with ligands: metabolites, drug compounds, DNA and others.

From molecular data we get information on coordinates and then we want to obtain some kind of knowledge, for example:

- Mutation X disrupts the function of enzyme Y which causes disease Z.

"Coordinates by themselves just specify shape and are not necessarily of intrinsic biological value, unless they can be related to other information"
Integrative database analysis in structural genomics, Mark Gerstein, Nature Structural Biology

There are many databases that store information on structural data for many proteins or other macromolecules, some of them also contains sequence data or metabolic pathways data. Sequence data are just data on DNA so the sequences, while metabolic pathways are related to which pathway a given gene activate.

Down below we can see a picture of most databases for bioinformatics data.

The first database for structural data was PDB (Protein Data Bank), with the purpose of building an archive on protein structural data. A lot of the newer databases for structural data are derived from PDB.

Lastly, we can see a figure representing the most used biological technologies for obtaining structural data.

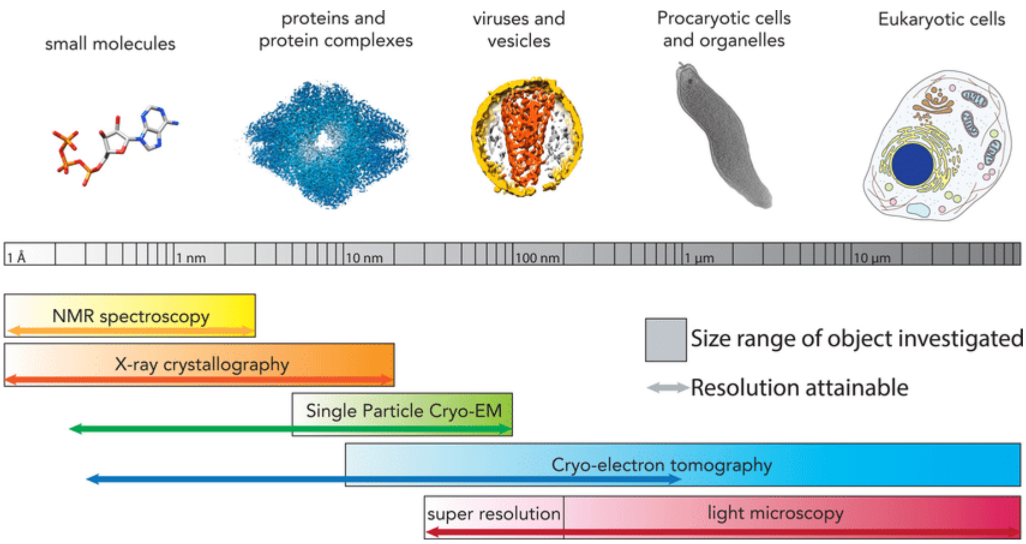


Figure 2.9: Biological technologies for obtaining structural data, from [6]

In our case, for protein chains and residues, the most interesting biological technologies are the first two from left:

- X-ray crystallography;
- NMR spectroscopy.

To close up this chapter I will talk about one of the most important libraries in bioinformatics: BioPython.

2.2.2 BioPython

Regarding Computer Science the main languages used for bioinformatics are:

- R;
- Matlab;
- Python.

For python there is the library BioPython, which is a library containing classes, functions and modules for the various necessities in bioinformatics.

Figure 2.10: BioPython logo

There are a lot of subpackages under the package Bio of BioPython, in our case we are just interested in 3 of them:

- **Bio.PDB**: contains classes that deal with macromolecular crystal structures. In particular it includes PDB and mmCIF parsers, the DSSP wrapper[2], the SASA module and the Structure class;
- **Bio.Data**: collection of useful biological data, for example in our case we use it for the normalization factors for RSA;
- **Bio.SeqUtils**: contains mhelper functions to deal with sequences.

In particular from Bio.PDB we use the classes:

- *PDBParser*: as the name suggests, it's a class which reads a .pdb file and saves it into a variable of the type Structure class;

[2] A wrapper is a class or procedure that translates a library's existing interface into a compatible interface, it "wraps" the underlying library. It's often used to enable cross-language, in this case the wrapper enables the c++ library "DSSP" to be used in Python.

- *MMCIFParser*: similar to PDBParser, but for .mmCIF files;
- *DSSP*: the python wrapper for the software tool DSSP;
- *SASA*: a class for calculating solvent accessibility;
- *Polypeptide*: a class with helper functions to deal with protein chains, in particular we use the function **is_aa()** to see if a string is an identifier for an amino acid.

On the other hand the Bio.Data subpackage has been used just for the normalization factors for SASA, the dictionary *residue_sasa_scales*.

And Bio.SeqUtils has been used for the helper function *Seq1*, which converts a sequence of amino acids with three-letters code into a sequence of amino acids with one-letter code.

For more information on BioPython and its subpackages and submodules, here is the link to the documentation.

2.2.3 DSSP

In BioPython we have an important procedure: the DSSP wrapper. DSSP (Define Secondary Structure Prediction) is a computer database used in structural bioinformatics to analyze a protein chain, in particular the secondary structure and other statistics that can be derived from it.

It uses hydrogen bond patterns and other geometric features of the protein to analyze it.

3 Analysis of AlphaFold-disorder

3.1 Introduction

3.1.1 AlphaFold

AlphaFold is a deep learning system developed by DeepMind. It's designed to predict protein folding, which is very important to study the protein functions. It uses deep networks to predict the three-dimensional structure of proteins.

Here the reference to the scientific paper [7].

3.1.2 pLDDT

pLDDT (per-residue Local Distance Difference Test) is a metric used by AlphaFold to indicate the confidence of predictions.

A high pLDDT score indicates higher confidence in the predicted 3D position of that residue in the protein structure. AlphaFold obtains it by computing the distances between the predicted position and the experimentally observed position of the residue.

3.1.3 Disorder

Disorder, in the context of Molecular Biology of proteins, refers to how "out-of-order" is the structure of a part of the protein, usually a region. Two of the three algorithms inside the software tool predict the disorder propensity of a certain residue, amino acid.

This disorder propensity is a probability for that residue of being part of a disordered region. Disordered regions are quite important to study, because, as we talked about in the previous chapter, proteins with intrinsically disordered regions are involved in many functions and pathologies.

3.1.4 Binding

The other algorithm within the software tool is a predictor for binding, which means it predicts a binding propensity for each residue, namely the probability for that residue to bind with some ligand.

This binding propensity can overlap with disorder propensity, since many disordered regions tend to have higher binding activity.

3.2 AlphaFold-disorder

AlphaFold-disorder is a software tool developed by Damiano Piovesan, Alexander Miguel Monzon and other members of the BioComputing UP laboratory. The objective of the software tool is to predict disorder and binding of the amino acids with high accuracy.

The CAID portal (Critical Assessment of Protein Intrinsic Disorder) [3], also developed by members of the BioComputing UP laboratory, was used to measure the quality of the software tool prediction.

In particular 3 algorithms within the tool were measured:

- **AlphaFold-pLDDT**: uses pLDDT to predict disorder values;
- **AlphaFold-rsa**: uses pLDDT and RSA statistics to predict disorder values;
- **AlphaFold-binding**: uses pLDDT and RSA statistics to predict binding values.

Each of these algorithms produces a table containing binding, disorder and other statistics for every amino acid, or residue[1].

The professors and researchers of the laboratory published a scientific paper[14] regarding this software tool: Intrinsic protein disorder and conditional folding in AlphaFoldDB.

[1]In molecular biology, a residue refers to a monomer within a polymeric chain. In the case of a protein, a residue refers to an aminoacid.

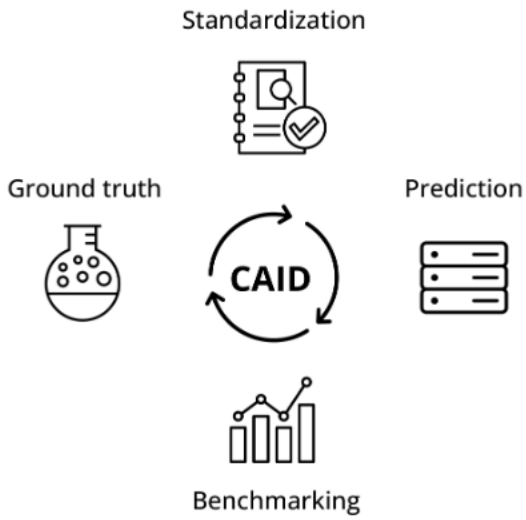


Figure 3.1: CAID cycle [3]

In this chapter we will see more in detail the CAID web application and the 3 algorithms within AlphaFold-disorder.

3.3 CAID

CAID, Critical Assessment of Protein Intrinsic Disorder prediction, is a web application established in 2018 to determine the state-of-art for predicting disorder and binding on IDRs, Intrinsic Disordered Regions.

The scientific paper for CAID portal is [3].

The idea is to compare different softwares for this kind of prediction: each of them assigns to every residue of the protein the propensity for that residue to be intrinsically disordered. Accuracy of predictions are evaluated by means of ground truths, obtained by a reference, in this case the reference is the DisProt database. On CAID both software runtimes and accuracy of predictions are compared.

Here we can see a picture representing the CAID cycle:

- **Ground Truth**: DisProt database was selected as reference because it contains a large number of manually curated disorder and binding annotations at a protein level. DisProt annotates association between intrinsically disordered regions and a biological function.
- **Standardization**: Participants submit their softwares, and the CAID organizers encapsulate code into containers, providing standardization across

different machines. This also makes it easier to deploy them and to package softwares with dependencies.

- **Prediction**: Running the containers implemented in the Standardization step on the Ground Truth targets generates predictions. The cluster used by CAID can execute all methods in parallel within some resource constraints (90 GB RAM, 48 threads, 4 Hours per sequence).

- **Benchmarking**: The performances of the various methods are evaluated by a number of metrics. CAID identifies the confidence threshold to optimize the performance for a specific metric. In benchmarking and runtimes pages are reported some of the assessment results.

The 3 algorithms we mentioned earlier, AlphaFold-pLDDT, AlphaFold-rsa and AlphaFold-binding, are already present in their containers on the CAID application.

3.4 AlphaFold-pLDDT

This predictor is provided by the software tool AlphaFold-disorder. It's a indicator of presence of disorder, it predicts the propensity for each residue to be intrinsically disordered.

Its predictions are based on the pLDDT, predicted Local Distance Difference Test, which is already present in .pdb files from AlphaFold predictions as the B-factor. This algorithm just obtains a statistic representing disorder propensity for that residue, out of the B-factor.

AlphaFold-pLDDT uses "1 - pLDDT" as disorder propensity.

The optimal classification threshold found with CAID is 0.312, representing pLDDT <68.8 %, this threshold was selected by maximizing the F1-Score performance on the CAID DisProt dataset.

3.5 AlphaFold-rsa

AlphaFold-rsa is another indicator of presence of disorder, within the software tool AlphaFold-disorder.

The disorder propensity is calculated with the relative solvent accessibility, RSA, over a local window centered on the residue whose disorder propensity we want to predict. DSSP was used to obtain the relative solvent accessibility and the optimal local window, of 25 residues, was chosen with a grid search in the range of 1 to 50 residues.

$$AlphaFold\_Bind(T) = \begin{cases} AlphaFold\_RSA, \\ \quad AlphaFold\_RSA \leq T \\ T + \rho LDDT(1-T), \\ \quad AlphaFold\_RSA > T \end{cases}$$

Figure 3.2: From AlphaFold paper [14]

The optimal classification threshold for AlphaFold-rsa is 0.581, which means RSA < 41.9 %. Again the threshold was selected by maximizing F1-Score performance on the CAID DisProt dataset.

3.6 AlphaFold-binding

Finally we have the last predictor in the software tool AlphaFold-disorder. This infer the binding propensity on disordered residues: disordered residues tend to have higher binding propensity (probability that residue has to bind with other molecules).

This predictor makes use of both the pLDDT and the RSA calculated for the other 2 predictors: in the scientific paper [14] that describes the software tool AlphaFold-disorder, we can see the relation between RSA, pLDDT and this binding propensity.

The parameter T represents a threshold, obtained again from maximizing the F1-Score on the CAID DisProt dataset. The optimal threshold found with CAID is 0.773.

Below 0.773 the binding propensity is the value from AlphaFold-rsa(T); above 0.773 we use the pLDDT score to compute the binding propensity.

3.7 Analysis of the Code

The main points are:

1. Parsing of input parameters;
2. Parsing of input file(s);
3. Extraction of pLDDT and RSA for each residue;
4. Computation of predictions;
5. Creation of output files.

3.7.1 Parsing Input Parameters

To parse input they used the library parse_args along with Path and PurePath of the library pathlib.

```
    import argparse
    from pathlib import Path, PurePath
```

Code 3.1: Import libraries parsing

The procedure "parse_args()" is used to parse parameters given in terminal, when using the software tool. These are the main parameters:

- **--in-struct (-i)**: the input file(s). Either a single file, folder or file listing containing the relative paths of .PDB or .mmCIF files;
- **--out (-o)**: name for the output file(s).
- **--format (-f)**: format for the output file, it can be either "tsv" or "caid";
- **-dssp**: the path for the mkdssp executable, by default is just the alias "mkdssp" which is needed to be setted in the environment variables;
- **-ll**: log level, it can be one of the following: "notset", "debug", "info", "warning", "error", "critical".

```
def parse_args():
    parent_parser = argparse.ArgumentParser(add_help=False)

    group = parent_parser.add_mutually_exclusive_group(required=True)
    group.add_argument('-i', '--in_struct', type=str,
            help='A single file, folder or file listing
```

```
            containing (gzipped) PDB or mmCIF files (relative paths)'
            )
[...]
    parent_parser.add_argument('-ll', type=str, choices=['notset',
        'debug','info','warning','error','critical'],
        default='info', help='Log level')
    main_parser = argparse.ArgumentParser(parents=[parent_parser])
    return main_parser.parse_args()
```

Code 3.2: Command-line arguments parsing.

3.7.2 Parsing Input Files

First of all the library pathlib is employed to obtain the path of the input file(s) and then we have 4 possibilities, identified through if-else 's :

- **one protein file**: so either a ".pdb" file, a ".pdb.gz", ".cif" or ".cif.gz" file;
- **list of files**;
- **a directory**: each protein file within this directory is processed;
- **non-protein files** are recognized and not processed.

```
if __name__ == '__main__':

[...]

    p = Path(args.in_struct)
    # args.in_struct is the CLI parameter for input files
    if p.is_file():
        # input is a single struct file or file with list
        if ''.join(PurePath(p).suffixes) in ['.pdb', '.pdb.gz',
        '.cif', '.cif.gz'] :
            #process single file as input
            processed_data = process_file(p)
[...]
        else:
            # process list of files as input
            with open(p, 'r') as list_file:
                for file in list_file:
                    real_file = Path(p.parent, Path(file.strip()))
                    processed_data = process_file(real_file)
[...]
    else:
```

```
        # input is a directory
        start_T = time.time()
        for file in p.iterdir():
            if ''.join(PurePath(p).suffixes) in ['.pdb', '.pdb.gz',
            '.cif', '.cif.gz'] :
                #process every file in the directory
                processed_data = process_file(p)
```

Code 3.3: Input files parsing

3.7.3 Extraction of Residues' Statistics

This part is implemented under the main procedure "process_pdb". To implement this procedure a few libraries have been used: the main ones are BioPython and pandas. The input parameters for the procedure are three:

- **pdb_file**: the actual protein file, so the path to that file;
- **pdb_name**: the name of the file;
- **dssp_path**: the path to the mkdssp executable.

Description of the procedure

The procedure can be divided in 4 steps :

1. **Load structure**: the protein file is loaded and parsed into a proper data structure: "Bio.PDB.Structure", a class in BioPython that represents a macromolecular structure. This parsing is done through "Bio.PDB.PDBParser" if it's a .pdb file or with "Bio.PDB.FastMMCIFParser" if it's a .mmCIF file.

2. **DSSP invocation**: DSSP is a software tool used to obtain interesting statistics for each residue. The software tool is invoked through a Python wrapper[2] class: "Bio.PDB.DSSP" provided by BioPython.

3. **Extraction of statistics**: Finally the statistics of interests are extracted by iterating the residues. The statistics of interests are 3:

[2]A wrapper is a class or procedure that translates a library's existing interface into a compatible interface, it "wraps" the underlying library. It's often used to enable cross-language, in this case the wrapper enables the c++ library "DSSP" to be used in Python.

1. **pLDDT**: obtainable without DSSP, from the B-factor value already present in the .pdb file;
2. **RSA**: it represents how much the solvent can access this residue. It's provided by DSSP;
3. **SS**: secondary structure of the residue. Also provided by DSSP.

4. **Save statistics in a DataFrame**: Finally the statistics of interest are stored in a DataFrame, a class provided by the library "pandas". Dataframe is the most popular way to manage a table and easily write it in a ".csv" or ".tsv" file.

Code snippet of the procedure

For more information on DSSP there is the original scientific paper [8] and the updated repository of github: DSSP. Down below we can see an adjusted code snippet to illustrate how this extraction of statistics for each residue works. It's adjusted for the purpose of showing it in this thesis.

```
import pandas as pd
from Bio.PDB import PDBParser, DSSP
from Bio.PDB.MMCIFParser import FastMMCIFParser
from Bio.SeqUtils import seq1
[...]
def process_pdb(pdb_file, pdb_name, dssp_path='mkdssp'):
[...]
    # Load the structure
    file_ext = Path(rel_file).suffix
    if file_ext in ['.pdb']:
        structure = PDBParser().get_structure('', real_file)
    else:
        # assume mmCIF
        structure = FastMMCIFParser().get_structure(''. real_file)
[...]
    # Calculate DSSP. WARNING Check the path of mkdssp
    dssp_dict = dict(DSSP(structure[0], real_file, dssp=dssp_path))
[...]
    # Parse b-factor (pLDDT) and DSSP statistics of interest
    df = []
    for i, residue in enumerate(structure.get_residues()):
        lddt = residue['CA'].get_bfactor() / 100.0
```

```
        rsa = float(dssp_dict.get((residue.get_full_id()[2],
residue.id))[3])
        ss = dssp_dict.get((residue.get_full_id()[2], residue.id))[2]
        df.append((pdb_name, i + 1, seql(residue.get_resname()),
lddt, 1 - lddt, rsa, ss))
    df = pd.DataFrame(df, columns=['name', 'pos', 'aa', 'lddt',
'disorder', 'rsa', 'ss'])
    return df
```

Code 3.4: Extraction residues' statistics

3.7.4 Computation of Predictions

We already have the predictions of AlphaFold-pLDDT, here we will describe the procedure for predictions of the other 2 algorithms: AlphaFold-RSA and AlphaFold-Binding.

The procedure is called *make_prediction* and it asks as input three parameters:

- **df**: dataframe containing the data on residue from DSSP;
- **window_rsa**: the window of residues to consider when calculating rsa;
- **thresholds_rsa**: the threshold of rsa that represents a high solvent-accessibility.

The last 2 parameters have a default value, based on empirical tests: 25 residues for the window_rsa and 0.581 for the thresholds_rsa.

The default value for window_rsa was found with grid search, with various numbers for the parameter, 25 was the best one. While for thresholds_rsa, the default value was obtained by maximizing the F1 score, on CAID.

Both in window_rsa and in thresholds_rsa we can have multiple values, for an easier comparison, the parameters are in fact lists of values.

By default this procedure adds to the dataframe 2 columns, one for AlphaFold-RSA disorder prediction and one for AlphaFold-Binding binding prediction. With more than elements in the parameter lists of window_rsa and thresholds_rsa we will have more columns.

Down below a code snippet of the procedure.

```
def make_prediction(df, window_rsa=[25], thresholds_rsa=[0.581]):
    for w in window_rsa:
        column_rsa_window='disorder-{}'.format(w)
        half_w = int((w-1)/2)
        tmp_pad = np.pad(df['rsa'], (half_w, half_w), 'reflect')
```

```
6         # running mean of array tmp_pad, with window: half_w*2 +1
7         df[column_rsa_window] = moving_average(tmp_pad, half_w*2 +1)
8 
9         for th_rsa in thresholds_rsa:
10            column_rsa_binding = 'binding-{}-{}'.format(w, th_rsa)
11            df[column_rsa_binding] = df[column_rsa_window].copy()
12            df.loc[df[column_rsa_window]>th_rsa,column_rsa_binding] =
13 df.loc[df[column_rsa_window]>th_rsa,'lddt'] * ((1-th_rsa)+th_rsa)
```

Code 3.5: Procedure make_prediction()

At line number 6 a procedure moving_average() is invoked: the procedure uses the NumPy library to compute the moving average of an array, using convolution.

While on lines 11 and 12 we have some complex computation that uses the thresholds_rsa to transform the scores.

3.7.5 Creation of Output Files

Finally the data obtained from DSSP and the predictions computed are saved in 2 .tsv files. Actually the first .tsv file is done before the procedure make_prediction, since it doesn't include the predictions' columns. For simplicity of explanation I preferred to talk about it in the same part as for the 2nd output file.

This creation of output files is incorporated in the main procedure, using the function "to_csv" provided by the Pandas library.

```
if __name__ == '__main__':
    fout_path = Path(args.out)
[...]
    data = data.append(processed_data)
[...]
    fout_name = '{}/{}_data.tsv'.format(fout_path.parent, fout_path.
   stem)
    # First .tsv file
    data.to_csv(fout_name, sep='\t', quoting=csv.QUOTE_NONE, index=
   False, float_format='%.3f')
[...]
    for name, pdb_data in data.groupby('name'):
        pred = pred.append(make_prediction(pdb_data.copy(),
   window_rsa=args.rsa_window,
            thresholds_rsa=args.rsa_threshold))
```

```
[...]
    fout_name = '{}/{}_pred.tsv'.format(fout_path.parent, fout_path.
   stem)
    pred.to_csv(fout_name, sep='\t', quoting=csv.QUOTE_NONE, index=
   False, float_format='%.3f')
```

Code 3.6: Creation of output files

fout_path is the information of name and path provided as parameter in the invocation of the software tool from shell, obtained form the parameter .out of the result from the output data structure of parse_args procedure, seen in Code Snippet 3.1. data is the DataFrame obtained from the procedure process_pdb, seen in Code Snippet 3.4. It contains the data from the protein files and their statistics obtained with DSSP. pred is the DataFrame obtained from the procedure make_prediction, seen in Code Snippet 3.5. It contains the predictions computed in that procedure and the data from protein files and DSSP.

name	pos	aa	lddt	disorder	rsa	ss	disorder-25	binding-25-0.581
109m	1	M	0.299	0.701	0.198	C	0.098	0.098
109m	2	V	0.254	0.746	0.362	C	0.089	0.089
109m	3	L	0.196	0.803	0.021	C	0.088	0.088
109m	4	S	0.171	0.829	0.086	C	0.09	0.09
109m	5	E	0.163	0.837	0.149	H	0.099	0.099
109m	6	G	0.159	0.841	0.031	H	0.092	0.092
109m	7	E	0.134	0.866	0.016	H	0.099	0.099
109m	8	W	0.127	0.873	0	H	0.109	0.109
109m	9	Q	0.123	0.877	0.169	H	0.109	0.109
109m	10	L	0.12	0.88	0.038	H	0.109	0.109
109m	11	V	0.11	0.89	0	H	0.109	0.109
109m	12	L	0.12	0.88	0.019	H	0.108	0.108
109m	13	H	0.12	0.88	0.239	H	0.093	0.093

Table 3.1: Columns of an AlphaFold-disorder output file

3.8 Usage of AlphaFold-disorder software tool

3.8.1 Dependencies

In order to use AlphaFold-disorder, we first need :

- the actual Python script **alphafold_disorder.py**;

- at least one protein file, in format **.pdb**, **.mmCIF**, **.pdb.gz** or **.mmCIF.gz**;
- the DSSP executable: **mkdssp**;
- the following libraries: **BioPython**, **Numpy** and **Pandas**.

3.8.2 Usage

To execute the software tool we can write the following command in the CLI:

```
python3 AlphaFold-disorder -i pdbs/ -o output
```

Instead of "output" we can write the name for the output file that we desire, and instead of "pdbs/" we can give as input a protein file, a directory containing protein files or a file with the relative paths of protein files.

3.8.3 DSSP Installation

There are two ways to install DSSP:

1. Download the pre-compiled file from the latest release in GitHub;
2. Build the the source file from the GitHub repository.

To build the source file we have to install and build a few libraries, but it's all well-document in the readme.md file of the DSSP repository.

3.8.4 Python Libraries Installation

To install the 2 Python libraries required we need **Python3** and **pip3**, Package Installer for Python. Then to install the 2 Python libraries we just have to execute the following commands from the CLI:

- **BioPython**: pip3 install biopython;
- **Numpy**: pip3 install numpy;
- **Pandas**: pip3 install pandas.

4

AlphaFold-disorder (SASA): Development of Procedures

We developed a variation of AlphaFold disorder: **AlphaFold-disorder (SASA)**. We developed a procedure to compute secondary structure of residues (PSEA algorithm) and a procedure to compute RSA, using the SASA algorithm.

In this chapter I will show the actual development part of the thesis. We worked on three different procedures:

- **Detection of secondary structure using protein atomic coordinates**: development of a procedure that assigns to each residue its secondary structure. It's based on P-SEA algorithm, described in this scientific paper[11];

- **Computation of RSA, using Shrake-Rupley algorithm**: We calculated SASA using Shrake-Rupley algorithm and then computed RSA with normalization factors;

- **Implementation of FoldComp library**: for reading protein files in .fcz, which is a compression of .pdb files.

4.1 Secondary Structure Detection with Protein Atomic Coordinates

4.1.1 Introduction

In the scientific paper of PSEA [11], the researchers suggest an algorithm based on protein atomic coordinates of the central carbon atom of the residue, the α-Carbon. This algorithm assigns the secondary structure to a residue r, based on distances and angles between the α-carbon of the residue and the α-carbon of its neighbor residues.

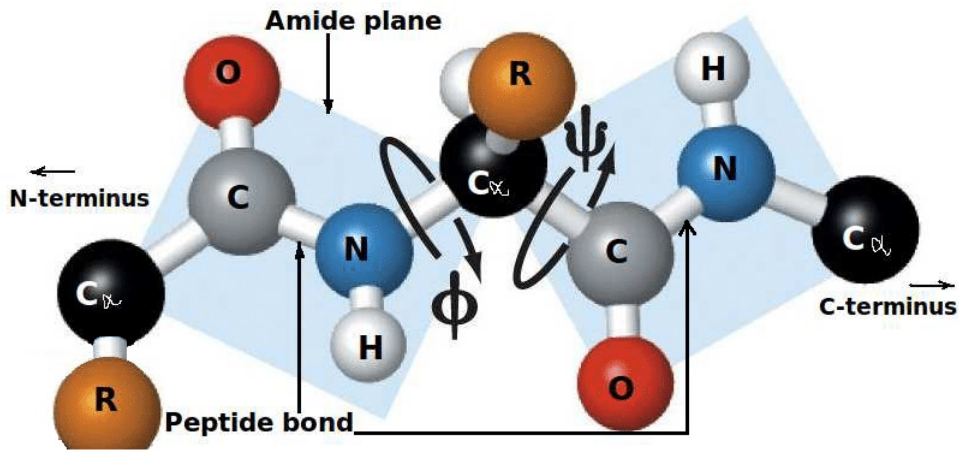


Figure 4.1: α-Carbon atoms in a protein, from [16]

α-Carbon atoms are the core of residues, in figure 4.1 we can see them in a piece of a protein chain.

Now back to the P-SEA algorithm: for each residue i, we want to calculate these distance measures:

- d2i: it's the distance between the residue (i-1) and the residue (i+1);
- d3i: it's the distance between the residue (i-1) and the residue (i+2);
- d4i: it's the distance between the residue (i-1) and the residue (i+3);

And these angle measures:

- τi: the angle formed by the residues (i-1), i and (i+1);
- αi: the dihedral angle formed by the residues (i-1), i, (i+1) and (i+2).

Distances and angles are computed between the α-carbons of bonded residues. Specific range of values in d2, d3, d4, τi, αi determine in which secondary structure that residue falls into, among the main three categories:

- **Helices**;
- **Strands**: such as beta-sheets parallel and anti-parallel;
- **Coils**: turns and loops;

The thresholds for belonging in one of the two non coil secondary structures are shown in table 4.1.

Parameters	Helix	Strand
Angle τ (°)	89 ± 12	124 ± 14
Dihedral angle α (°)	50 ± 20	-170 ± 45
Distance d2 (Å)	5.5 ± 0.5	6.7 ± 0.6
Distance d3 (Å)	5.3 ± 0.5	9.9 ± 0.9
Distance d4 (Å)	6.4 ± 0.6	12.4 ± 1.1

Table 4.1: Parameters for P-SEA assignment of secondary structure.

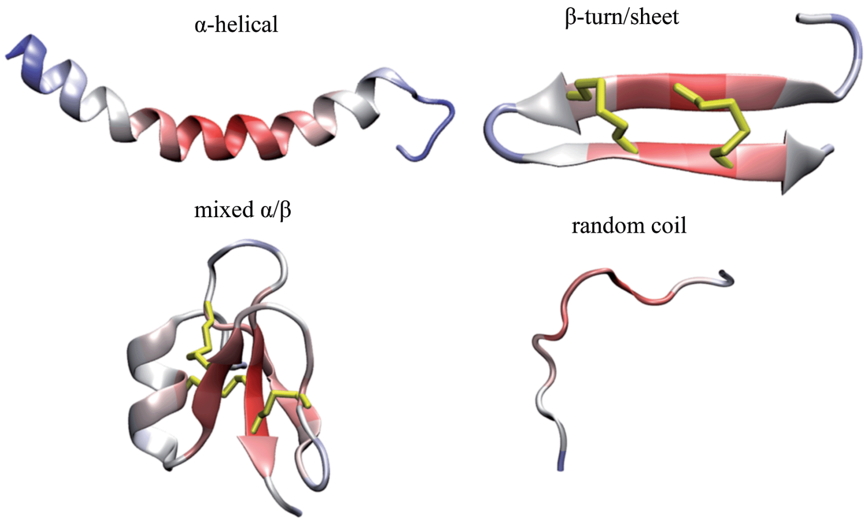


Figure 4.2: The three secondary structure P-SEA can identify, from [15]

In the next section I will show the implementation of the PSEA algorithm discussed on the paper[11].

4.1.2 Implementation of PSEA procedure

We can summarize the PSEA algorithm in these steps:

1. Get the coordinates of the residue's α-carbon atom, for each residue in the protein;
2. Compute distances and angles for each residue, as described in the paper;
3. Assign to each residue its secondary structure, based on computed distances and angles.

Get α-carbon atom's coordinates for each residue

In this first step we want to obtain the list of residues in the protein and then for each residue we want the coordinates of its α-carbon.

We store them in a matrix with 3 columns and as many rows as the number of residues. In each row we store the 3 coordinates of the α-carbon atom.

The basic idea would be to:

- Obtain the list of atoms from the protein chain;
- Obtain the list of α-carbon atoms from the whole list of atoms;
- For each α-carbon atom, obtain its residue with the BioPython function "get_parent()";
- Store the coordinates of that α-carbon atom in the row of that residue, in the matrix.

This is the basic pseudo-code for this step. Although internally, the code is more optimized for computational purposes: we used NumPy methods to use the residues starting indexes more efficiently to create the α-carbon coordinates' matrix.

Algorithm 1 Pseudocode for extracting α-carbons coordinates

```
Require: struct, a protein structure
  list_atoms = struct.get_atoms()
  res_start_ids = get_res_start()
  list_ca = empty list of size "len(res_start_ids)"
  for atom in list_atoms do
    if atom.name = "CA" then
      list_ca.append(atom)
    end if
  end for
  matrix_coordinates = empty matrix of size "len(res_start_ids) x 3" {Res start
  ids are long the number of residues in the protein chain, and 3 because we
  have 3 coordinates: x,y,z.}
  for ca in list_ca do
    matrix_coordinates.append(ca.coordinates)
  end for
```

Compute distances and angles between α-carbon atoms

Now we have a list of α-carbon atoms, with their coordinates. The next step described in the paper is computing distances and angles between these atoms, to assign the correct secondary structure.

For each row of the matrix, which represents the residue's α-carbon atom, compute the 5 measures:

- **d2**: distance between atom at row i and atom at row i+2;
- **d3**: distance between atom at row i and atom at row i+3;
- **d4**: distance between atom at row i and atom at row i+4;
- angle τ: angle between three consecutive atoms: at rows i, i+1 and i+2;
- angle α: angle between four consecutive atoms: at rows i, i+1, i+2 and i+3.

I will show with a few pictures what distances and those 2 angles looks like, in the case of 5 consecutive α-carbons (α-carbon are directly linked for simplicity).

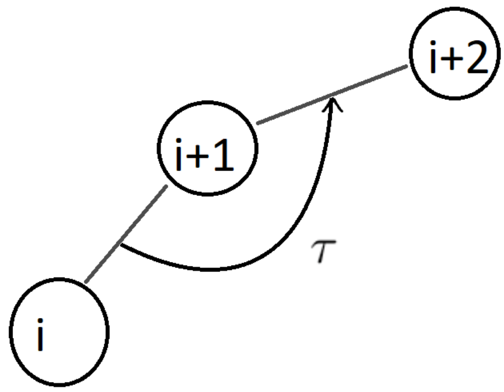


Figure 4.3: Angle τ

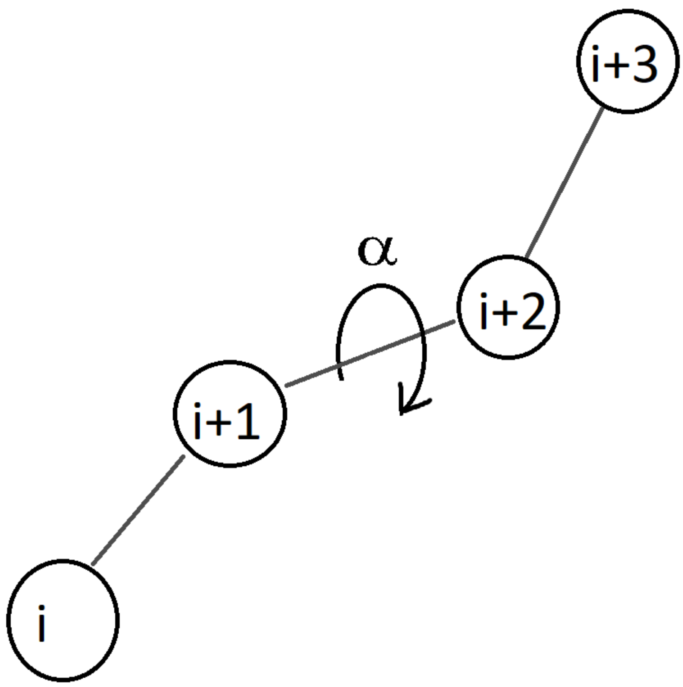


Figure 4.4: Dihedral Angle α

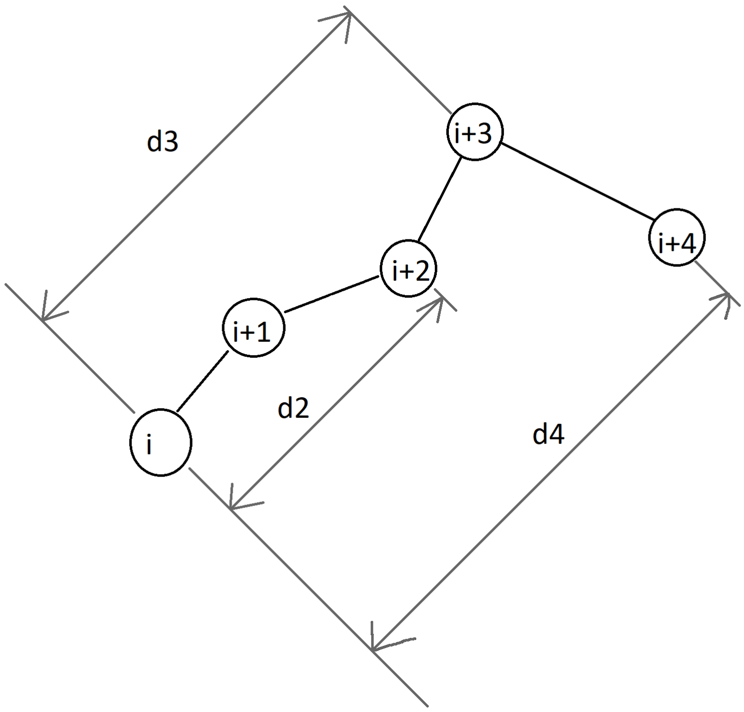


Figure 4.5: Distances d2, d3 and d4

Measures:

- **Distances**: for the distances we just use the euclidean norm, which is $||x||_2 = \sqrt{x_1^2 + ... + x_n^2}$
- **Angle** τ: for this angle we used the following formula $\tau = arccos(\frac{a \cdot b}{||a|| \cdot ||b||})$. With $a = (v_{i+1} - v_i)$ and $b = (v_{i+1} - v_{i+2})$.
- **Angle** α: to calculate the dihedral angle between the 4 data points we have to calculate the angle between the two half-planes defined by three consecutive points. The first semi-plane n_1 is defined by the points v_i, v_{i+1} and v_{i+2}. The second semi-plane is defined by the points v_{i+1}, v_{i+2} and v_{i+3}.

1. We computed the vectors u,v and w by vector subtraction: $a = \frac{v_{i+1}-v_i}{||v_{i+1}-v_i||}$, $b = \frac{v_{i+2}-v_{i+1}}{||v_{i+2}-v_{i+1}||}$ and $c = \frac{v_{i+3}-v_{i+2}}{||v_{i+3}-v_{i+2}||}$;
2. We computed the two half-planes $n_1 = a \times b$ and $n_2 = b \times c$ [1];
3. Finally we obtain the angle as $\alpha = arctan2((n_1 \times n_2) \cdot b,\ n_1 \cdot n_2)$.

Here we can see the pseudo-code that computes distances and angles.

Algorithm 2 Pseudocode for computing angles and distances between α-carbons coordinates

Require: ca_coord, a matrix containing the coordinates of the various α-carbon atoms

```
d2 = distance_atoms(ca_coords[:-2], ca_coords[2:])
d3 = distance_atoms(ca_coords[:-3], ca_coords[3:])
d4 = distance_atoms(ca_coords[:-4], ca_coords[4:])
tau = angle(ca_coords[:-2], ca_coords[1:-1], ca_coords[2:])
alpha = dihedral(ca_coords[:-3],ca_coords[1:-2],ca_coords[2:-1],ca_coords[3:-2])
```

As we can see from the pseudo-code we used the python language built-in functions to compute all distances and angles in oneshot, but it's possible to implement it with for loops.

Assign secondary structure to each residue

For this last step, we want to compute which residue belongs to a helix structure and which residue belongs to a strand structure. The paper describes the procedure to assign to a residue the secondary structure, based on distances and angles, using the table for criteria: table 4.1.

For assigning residues to helical structure:

- Helices are first assigned to any segment at least five residues long, satisfying either one of the following helical criteria:
    1. Each residue's α-carbon in the segment satisfies both the helical criteria for d3i and for d4i distances;
    2. Each residue's α-carbon in the segment satisfies both the helical criteria for αi and τi angles.

[1]This $\times$ symbol is to indicate the cross product between two vectors, while the $\cdot$ symbol is for dot product. More information on the cross product here.

- Then each of this segment is lengthened by one residue at each end and if those 2 residues satisfy some criteria, we assign them to helix structure too. The criteria is that the residue satisfies either:

    1. d3i distance helical criteria;
    2. τi angle helical criteria.

Regarding the assignment of strand structure to residues, the procedure is similar:

- Strand structure is assigned to any segment at least three residues long, satisfying either one of the following strand criteria:

    1. Each residue's α-carbon in the segment satisfies both the d2i, d3i and d4i distances criteria;
    2. Each residue's α-carbon in the segment satisfies both the τi and αi angles criteria.

- Then lengthening by one residue at each end if it satisfies the d3i distance strand criteria;

- Finally short β-strand (<4 residues) are kept only if they have enough contacts, which means they are included in a β-sheet.

Algorithm 3 Pseudocode for assigning secondary structure to residues

Require: ca_coords, d2, d3, d4, α, τ
  basic_helix = (d3 $\in$ (4.8, 5.8) AND d4 $\in$ (4.8, 7)) OR (τ $\in$ (75, 101) AND α $\in$ (30, 70))
  basic_helix = **mask_consecutive**(basic_helix, 5)
  extended_helix = **extend_region**(basic_helix)
  basic_strand = (d3 $\in$ (4.8, 5.8) AND d4 $\in$ (4.8, 7)) OR (τ $\in$ (75, 101) AND α $\in$ (30, 70))
  basic_strand = **mask_consecutive**(basic_strand, 3)
  extended_strand = **extend_region**(basic_helix)
  extended_strand = **mask_regions_with_low_contacts**(extended_strand)
  ss_prediction = list long len(ca_coords) initialized with all "C"
  ss_prediction[extended_helix] = "H"
  ss_prediction[extended_strand] = "E"

The procedures used in this pseudocode will be described later on in their section 4.1.3:

- **extend_region**;
- **mask_consecutive**;
- **mask_regions_with_low_contacts**.

After this step every residue will have its secondary structure assigned:

- **H** for helix structure residues;
- **E** for strand structure residues;
- **C** for coils;
- **"** for residues without the α-carbon.

4.1.3 Helper procedures

In this section we will describe more in detail the helper procedures used for the PSEA algorithm:

- **get_res_start()**: procedure that creates a list of atoms out of the protein structure and then computes the start of each residue, by changes of residue name, id or chain id. It's possible to pass as parameter to the procedure "move_hhem_to_end = True", this puts those atoms between chains to the end of the atom array, resulting in a higher accuracy of prediction of secondary structure in multi-chain proteins.
- **distance_atoms()**: computes the distance between two atoms;
- **angle()**: computes the angle between three atoms;
- **dihedral()**: computes the dihedral angle between four atoms, between their semiplanes;
- **mask_consecutive()**: this procedure is used to create the first step of helix and strand mask, where we want at least a segment of 3/5 residues satisfying a certain condition. It uses binary masks along with the library NumPy;
- **extend_region()**: this procedure is for the second step of finding if a residue belongs to that structure, which is lengthening. This procedure lengthens the segment if the residues satisfy the necessary criteria;
- **mask_regions_with_low_contacts()**: finally this procedure is used in the strand structure mask, short strands are kept only if they have enough contacts. This procedure removes short strands with low contacts from the strand masks.

I decided to not show the code snippets for these procedures because they are complex and not too useful to understand the code, but they can be found in the github repository in the file "utils_alphafold_disorder.py".

4.1.4 Integration

For the integration with the existing AlphaFold-disorder software tool, we encapsulated this PSEA algorithm in a procedure called get_sse_psea() which takes as input the protein structure and returns as output an array of characters, with the secondary structure of each residue. The procedure called from the software-tool for each protein file is called "process_pdb_psea()". Based on which method we want to use there is a "process_pdb_psea()" procedure and a "procedure_dssp()" one.

So the existing code isn't too much touched, since the only difference is that it has to call this procedure instead of the DSSP wrapper procedure.

Down below a code snippet of this procedure encapsulating the whole PSEA algorithm.

```
def get_sse_psea(structure, add_short_contacts = True, move_end_hhem
    = True) :
    res_start_id, atom_array = get_res_start(structure,
        move_end_hhem)
    ca_coord, novirtual_mask = get_ca_coord(res_start_id, atom_array)
    length, [d2i, d3i, d4i, ri, ai] = calc_dist_angles(ca_coord)
    helix_mask, strand_mask = calc_struct_mask(ca_coord, length,
        [d2i, d3i, d4i], [ri, ai], add_short_contacts)

    return finalize_sse(ca_coord, length, novirtual_mask,
        helix_mask, strand_mask)
```

Code 4.1: Procedure get_sse_psea()

We already talked about the procedures **get_res_start**, while **get_ca_coord**, like the name suggests creates the matrix of α-carbon coordinates, **calc_dist_angles** is the procedure for computing the distances and angles. **calc_struct_mask** is the procedure to compute the helix mask and strand mask, this procedure takes as parameter "add_short_contacts", which if set to true removes short strands with low contacts.

And finally the **finalize_sse** procedure uses the helix and strand masks to assign the secondary structure to each residue and returns an array of characters, each character representing the secondary structure for the residue at that array position.

We compared the secondary structure predicted with this method and the secondary structure predicted with DSSP for a few hundred proteins and as the paper[11] of PSEA said we got an accuracy of around 80%.

To compare that we used a procedure called "compare_sses" which compares the secondary structure elements predicted from DSSP with the secondary structure predicted with PSEA algorithm.

```
def compare_sses(sse, dssp) :
    sse_dssp = [dssp[i][2] for i in dssp]

    counter1 = 0
    counter2 = 0
    countersize = 0
    for (i,j) in zip(sse, sse_dssp) :
        if j in ['H','G','I'] :
            j = 'a'
        elif j in ['B', 'E'] :
            j = 'b'
        else :
            j = 'c'
        if i == '': i = 'c'

        if i == j :
            counter1 += 1
        if i in ['a','b'] : i = 'p'
        if j in ['a','b'] : j = 'p'
        if i == j :
            counter2 += 1
        #else :
            #print(i,j)
        countersize+=1
    counter1 = counter1/countersize
    counter2 = counter2/countersize
    if counter1 == counter2 :
        return round(counter1,4)
    else :
        return [counter1/countersize, counter2/countersize]
```

Code 4.2: Procedure to compare SSEs

4.2 Computation of RSA with Shrake-Rupley Algorithm

We implemented an alternative way to compute the RSA, without the DSSP. This method is uses the Shrake-Rupley algorithm, to compute SASA, solvent accessible surface areas. This value is then transformed into the relative solvent accessibility with two rounds of normalization factors: first by using Sander's normalization factors, like in DSSP, and then by using an ad-hoc normalization factors we made to not have rsa values bigger than 1, obtained from a sample of two thousands proteins.

We can summarize the steps for implementing this procedure in 2 steps:

- Implement Shrake-Rupley algorithm to obtain the SASA statistics, for each residue;
- Normalization to obtain RSA statistics: both with Sander and Rost's normalization factors, explained in their scientific paper [19] and with our ad-hoc normalization factors.

4.2.1 Implement Shrake-Rupley algorithm

The Shrake & Rupley algorithm consists in a "rolling ball" of radius equal to a solvent molecule, which estimates the surface of the target molecule. This algorithm allows us to compute the SASA statistic (Solvent Accessible Surface Areas), more details can be found in the relative scientific paper[21].

To implement this algorithm in our code we just imported the library "Bio.PDB.SASA" and used the provided class "ShrakeRupley", with its function "compute()". We used the class to create an object of type "ShrakeRupley" and then applied the function "compute()" passing as parameters the protein structure and the level "R", R stands for residue, so that we compute the SASA statistic for each residue. We used the default paramters for the class ShrakeRupley since they are the right parameters for water as solvent, but in case of a different solvent we might want to change them.

We will see a snippet of the code for this procedure in the section 4.2.3.

4.2.2 Normalization

These SASA values aren't relative at all, they can have really high values, up to 200 or more. We want to obtain a statistic that represents the relative surface accessibility, so we have to perform some kind of normalization. We applied two rounds of normalization factors:

- **Sander and Rost's normalization factors**: these are factors for normalization developed by Sander and Rost, with the maximum values of ASA values (Accessible Surface Areas). To use them we import the library "Bio.Data.PDBData" which contains the dictionary "residue_sasa_scales", this dictionary provide the normalization factor for each residue, so we apply these normalization factors to the SASA value we obtained for each residue;

- **Ad-hoc normalization factors**: since after applying the Sander and Rost's normalization factors we still had RSA values bigger than 1, we designed another round of normalization factors. We took all the proteins from the DisProt database, which are 2547 proteins, computed the SASA values for each residue and divided them by the Sander and Rost's normalization factors obtaining the norm_SASA values. With all these norm_SASA values we built up a dictionary of ad-hoc normalization factors. Finally we divided the norm_SASA values by these ad-hoc normalization factors and obtained the final RSA values.

After these 2 rounds of division by normalization factors we obtain the RSA values, within a range of [0,1], representing the percentage of surface accessibility for that residue.

Now for reference I will show the values of the 2 rounds of normalization factors as tables.

3 Letters AA	**1 Letter AA**	**Normalization Factor**
Ala	A	106
Arg	R	248
Asn	N	157
Asp	D	163
Cys	C	135
Gln	Q	198
Glu	E	194
Gly	G	84
His	H	184
Ile	I	169
Leu	L	164
Lys	K	205
Met	M	188
Phe	F	197
Pro	P	136
Ser	S	130
Thr	T	142
Trp	W	227
Tyr	Y	222
Val	V	142

Table 4.2: Sander and Rost's normalization factors

3 Letters AA	**1 Letter AA**	**Normalization Factor**
Ala	A	1.713
Arg	R	1.280
Asn	N	1.476
Asp	D	1.435
Cys	C	1.551
Gln	Q	1.311
Glu	E	1.360
Gly	G	1.804
His	H	1.404
Ile	I	1.413
Leu	L	1.532
Lys	K	1.357
Met	M	1.420
Phe	F	1.434
Pro	P	1.490
Ser	S	1.515
Thr	T	1.507
Trp	W	1.423
Tyr	Y	1.346
Val	V	1.549

Table 4.3: Ad-hoc normalization factors

4.2.3 Integration

Regarding the integration into the exisiting software-tool AlphaFold-disorder, we implented this algorithm for RSA inside the procedure "process_pdb_psea" since it's an alternative way to compute rsa compared to the one in "process_pdb_dssp".

For the normalization factors:

- Sander's normalization factors: it's enough to import the dictionary "residue_sasa_scales" from "Bio.Data.PDBData", which contains the normalization factors of each residue;

- Ad-hoc normalization factors: we created a file .json called "dict_max_rsa.json", which contains the normalization factors of each residue.

Once we have the dictionary of the normalization factors, we can obtain the factor of our residue simply by obtaining the value paired with that residue in the dictionary.

Down below a code snippet of the implementation of this Shrake & Rupley algorithm to obtain the RSA values.

```
def process_pdb_psea(pdb_file, pdb_name) :
    structure, real_file = extract_pdb(pdb_file)
    with open('dict_max_rsa.json','r') as f :
        dict_adhoc_factors = json.load(f)
[...]
    sse = get_sse_psea(structure)
[...]
    ShrakeRupley().compute(structure, level="R")
    for i, residue in enumerate(structe.get_residues()):
[...]
        rsa = residue.sasa / residue_sasa_scales['Sander'][residue.
   resname]
        rsa = rsa / dict_adhoc_factors[protein_letters_3to1[residue.
   resname]]
[...]
```

Code 4.3: Integration of rsa with SASA on process_pdb_psea procedure

4.3 Implementation of FoldComp Library

We used the library FoldComp, described in more detail in their scientific paper [10], to allow to use a new input file in the software tool.

With FoldComp we can read and write .fcz file, which are compressed protein files, this enable us to store more protein files in the same storage space.

Regarding the implementation, we have:

- "**decompress**" procedure of FoldComp;
- we made a procedure for checking if a file is a .fcz file "**is_fcz_file**": if the file starts with *b'FCMP:'*, then it's a .fcz file.

Since in our case we are interested just in reading files .fcz we just need the procedure "decompress" from the library, which decompress the .fcz file into a .pdb file.

We used the library **StringIO** as a workaround to avoid saving the protein file after decompressing the .fcz file.

```
def is_fcz_file(filepath) :
    with open(filepath, 'rb') as test_f:
        return test_f.read(5) == b'FCMP:'

def extract_pdb(pdb_file) :
[...]
    elif is_fcz_file(real_file):
        with open(real_file, 'rb') as f :
            (name, pdb_filevalue) = foldcomp.decompress(f)
        structure = PDBParser(QUIET=True).get_structure('', StringIO(
   pdb_filevalue))
[...]
```

Code 4.4: Integration of FoldComp

5

AlphaFold-disorder (SASA) assessment

AlphaFold-disorder (SASA) is the variation we developed of the software-tool AlphaFold-disorder, using PSEA and SASA instead of DSSP to compute the predictions, in particular we have the SASA variation for each of the three algorithms within the software-tool:

- AlphaFold-pLDDT (SASA);
- AlphaFold-rsa (SASA);
- AlphaFold-binding (SASA).

Now we have to understand if this new software-tool produces good predictions: we need to evaluate the results' quality. To do so, we will analyse the output datasets, which contain the predictions, of AlphaFold-disorder and AlphaFold-disorder (SASA) and compare them to a ground truth:

- Analyse the columns of the output datasets;
- Create plots or visualizations, to gain insights on the results and patterns between features (statistics and predictions);
- Finally create ML models and analyse the statistics and main plots of the best models.

5.1 Output dataset

The software-tool produces two output files:

- output_data.tsv: contains the statistics before computing the predictions;
- output_pred.tsv: contains both the statistics and the predictions.

We will consider only the latter one, since it includes the first one inside. The name of the files depends on the parameter given by command line when using the software-tool.

Both in AlphaFold-disorder dataset and AlphaFold-disorder (SASA) dataset we have the same columns, an example of such a dataset in table 3.1.

5.2 Exploratory Plots

We produced some plots to explore the distributions of the various features and visualize their trends.

We obtained the ground truth from the CAID portal, and we considered the structural features of the proteins present in these ground truths. These ground truths are lists of "0", "-" and "1" for each residue in some proteins: respectively absent, present and uncertain. We described the procedure to obtain and process the CAID ground truths in 5.5.

The features we considered are:

- LDDT;
- RSA;
- disorder-25;
- binding-25.

Disorder-25 and Binding-25 are respectively the predictions of AlphaFold-rsa and AlphaFold-binding.

We have two ground truths for disorder and one for binding, we made plots per class: positive disorder and negative, and for binding too. We made density plots, histograms and scatterplots, using the Python library seaborn.

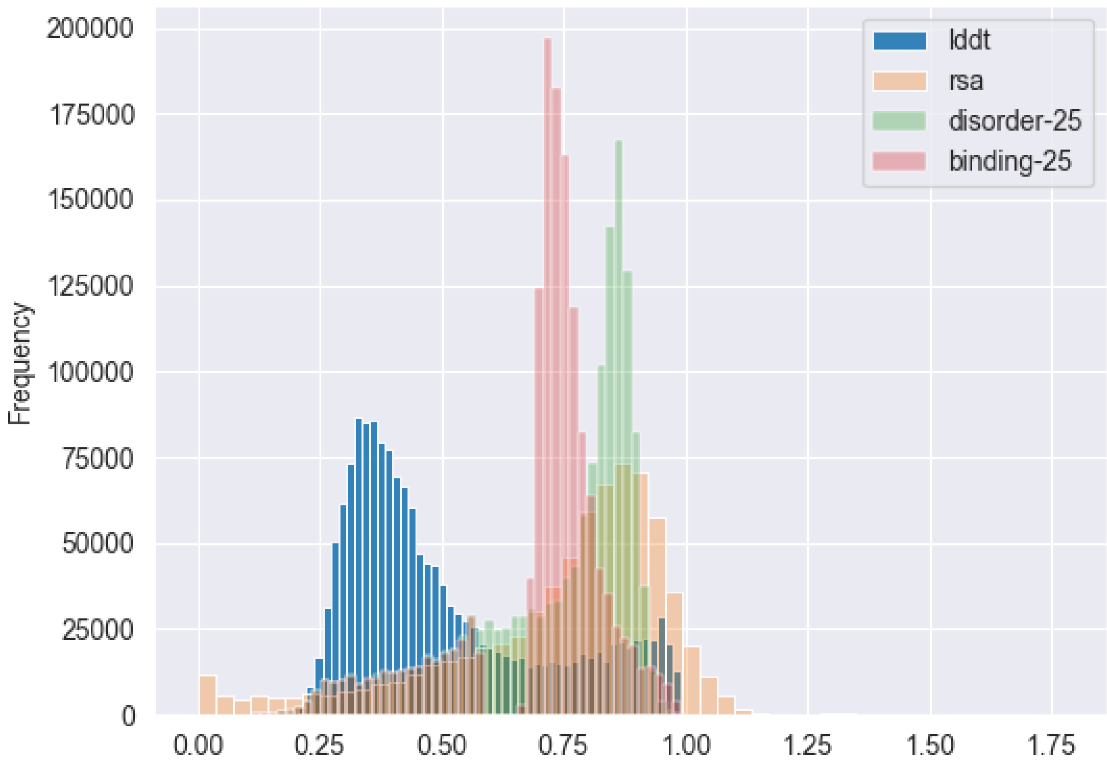


Figure 5.1: Histogram of the features considered, with respect to disorder ground truth (positive class)

In figure 5.1 shows the distribution of the features considered in this analysis, for residues with disorder, according to the ground truth.

In figure 5.2 we have another histogram to show distribution of features, this time on both classes of disorder (present - 1 vs absent - 0).

Another kind of plot we made are scatterplots: we made them between all pairs of the four considered features, to visualize any possible trend, in particular trends related with frequency.

I will show the scatter plots of the pair (rsa, lddt) for binding ground truth.

To further explore the results, we decided to develop some ML models and compare their Precision-Recall and ROC plots to the benchmark ones (AlphaFold-plddt, AlphaFold-rsa and AlphaFold-binding).

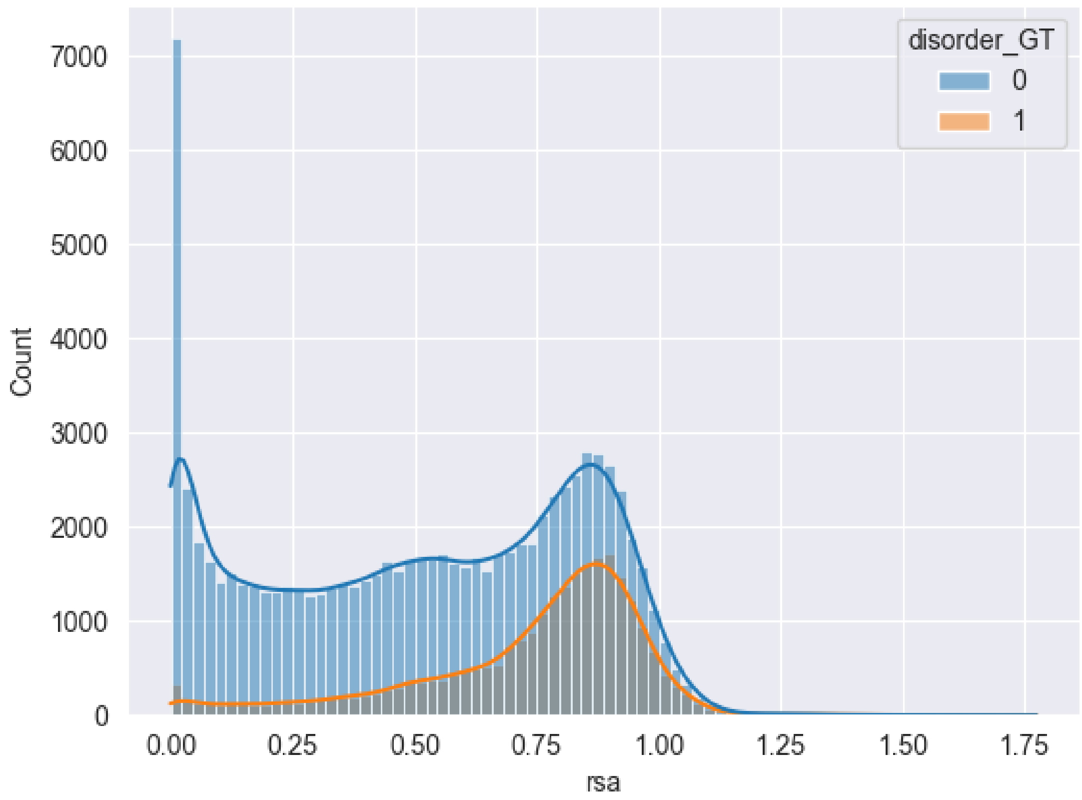


Figure 5.2: Histogram of the RSA feature, with respect to disorder ground truth (both classes)

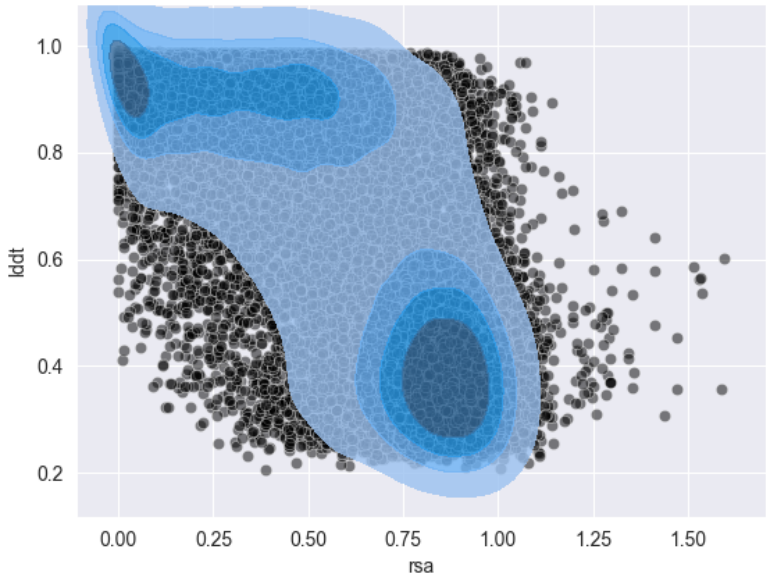
5.3 ML MODELS

For the machine learning models we considered two different ground truths for disorder and one for binding:

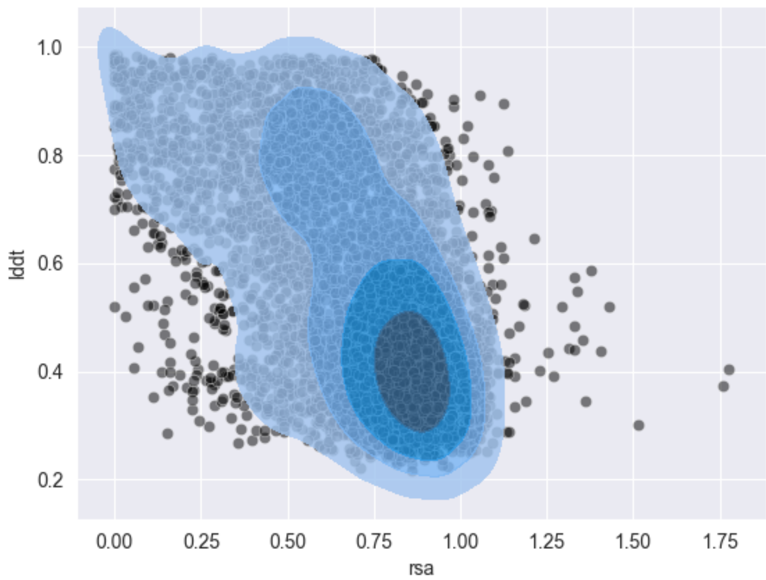
- disorder-pdb;
- disorder-nox;
- binding.

These three ground truths are from the CAID portal [3]. We described the procedure to obtain and process the CAID ground truths in 5.5.

We developed machine learning models using the three ground truths as target features for the test datasets: so we will have three test datasets, based on the ground truth used and one train dataset, where the target feature will be either binding or disorder, based on the test dataset used.



(a) Scatterplot (rsa, lddt) - positive class



(b) Scatterplot (rsa, lddt) - negative class

5.3.1 Preprocessing

As test datasets we used the dataframes containing the proteins present in each of the ground truths, with target the ground truths.

While as train dataset we used the dataframe of all the proteins in the DisProt database, using as target either the disorder or binding functional annotations present in DisProt, based on the test dataset used. We described the procedure to obtain these proteins and these functional annotations from DisProt in 5.5.

We computed a one-hot encoding of the ss column: **ss_onehot**, so we have a number instead of characters "H", "E", "C". We did this both for test datasets and for train datasets.

We also added new features to improve the accuracy of ML models: we incorporated the Atchley scale in the datasets. This Atchley scale consists in five features that are defined for each residue.

Amino acid	Factor I	Factor II	Factor III	Factor IV	Factor V
A	-0.591	-1.302	-0.733	1.570	-0.146
C	-1.343	0.465	-0.862	-1.020	-0.255
D	1.050	0.302	-3.656	-0.259	-3.242
E	1.357	-1.453	1.477	0.113	-0.837
F	-1.006	-0.590	1.891	-0.397	0.412
G	-0.384	1.652	1.330	1.045	2.064
H	0.336	-0.417	-1.673	-1.474	-0.078
I	-1.239	-0.547	2.131	0.393	0.816
K	1.831	-0.561	0.533	-0.277	1.648
L	-1.019	-0.987	-1.505	1.266	-0.912
M	-0.663	-1.524	2.219	-1.005	1.212
N	0.945	0.828	1.299	-0.169	0.933
P	0.189	2.081	-1.628	0.421	-1.392
Q	0.931	-0.179	-3.005	-0.503	-1.853
R	1.538	-0.055	1.502	0.440	2.897
S	-0.228	1.399	-4.760	0.670	-2.647
T	-0.032	0.326	2.213	0.908	1.313
V	-1.337	-0.279	-0.544	1.242	-1.262
W	-0.595	0.009	0.672	-2.128	-0.184
Y	0.260	0.830	3.097	-0.838	1.512

Table 5.1: Atchley scale

As the last step of preprocessing, we defined which features do we want to consider in the learning process and which one is the target.

For both the train and test datasets we consider as input features:

- lddt;
- rsa;
- ss_onehot;
- the five factors from the Atchley scale.

For the train datasets we consider as target feature the DisProt functional annotation, either binding or disorder annotation. In case we used as test dataset the one with target **binding ground truth**, we will use as target feature for the train set the binding functional annotations of DisProt, otherwise we will use as target feature the disorder functional annotations.

For the test datasets we consider as target feature either of the three ground truths.

5.3.2 Development of Machine Learning Models

For the development of the ML models we implemented the library **sklearn**, further details on the website.

The ML task we considered is regression, because we want to obtain the propensity of disorder or binding of the residues.

We implemented eight different ML models: svm, ridge, lasso, poisson-glm, tweedie, k-nearest neighbors, decision tree and extra tree.

In general to implement the libraries and compute the ML models we used the procedures provided by the library, in particular:

- fit(): this procedure allows us to fit the model with our train dataset;
- predict(): this procedure allows us to predict a value, given the input data of the test dataset. Then we will compare this value with the target value of the test dataset and evaluate the ML model performances.

We also have a class contructor to initialize each model, always provided by the library.

In the code snippets of the various ML models we will see the variables:

- **X_train**: input features of the train dataset;
- **y_train**: target feature of the train dataset;
- **X_test**: input features of the test dataset;
- **y_pred**: predicted values for output feature, given X_test.

Now I will show the different ML models and how we implemented them with the **sklearn** library.

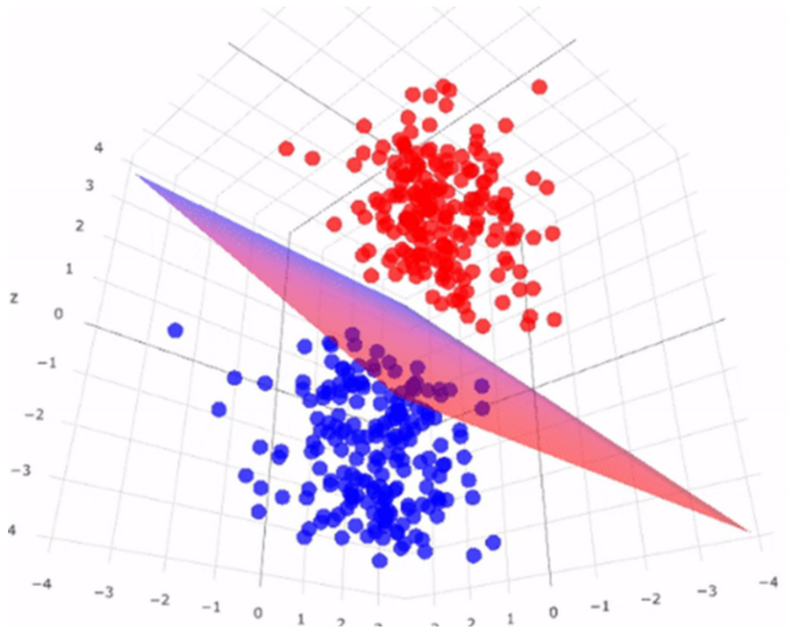


Figure 5.4: Picture of SVM hyperplane, from [17]

SVM

Support Vector Machines are one of the most simple yet powerful machine learning models. The objective is find an N-dimensional hyperplane that clearly separates data points of different classes.

Down below a code snippet of its implementation on a Python notebook.

```
from sklearn.linear_model import *
[...]
clf = SGDRegressor()
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
```

Code 5.1: SVM implementation

The class SGDRegressor is defined in the module **linear_model** of sklearn, SGDRegressor stands for Stochastic Gradient Descent because the SVM hyperplane is computed with stochastic gradient descent. This is the most basic linear model for regression.

5.3.3 Ridge

Ridge Regression, also known as Tikhonov regularization or L2 regularization, is a variation of linear regression: it adds a regularization term to make the model more robust.

The L2 regularization discourages large coefficient values: the larger the coefficient the more the regularization term penalizes the model. This makes the model more simple and stable, penalizing complex models.

```
from sklearn.linear_model import *
[...]
clf = RidgeCVRegressor()
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
```

Code 5.2: Ridge implementation

Here we used the **RidgeCV** class instead of simply **Ridge**, because we have cross-validation included too, which might increase performances. RidgeCV is also inside the module linear_model of sklearn.

5.3.4 LASSO

LASSO, Least Absolute Shrinkage and Selection Operator, is a linear regression model which uses L1 regularization. L1 regularization has the effect of setting some of the coefficients to exactly zero. This has the interesting property of performing feature selection, meaning that LASSO can be used to automatically select a subset of the most relevant features.

This is the ML model with better performances with our datasets, probably this automatic feature selection is very effective in our case.

```
from sklearn.linear_model import *
[...]
clf = LassoCV()
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
```

Code 5.3: LASSO implementation

LassoCV is also inside the module linear_model of sklearn. Even here the cross-validation is included.

5.3.5 Poisson GLM

Poisson Generalized Linear Model is a statistical model used for analyzing count data, where the data follows a Poisson distribution.

```
from sklearn.linear_model import *
[...]
clf = PoissonRegressor()
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
```

Code 5.4: Poisson-GLM implementation

PoissonRegressor is also inside the module linear_model of sklearn.

5.3.6 Tweedie GLM

Tweedie Generalized Linear Model is a statistical model used modeling data that follows a Tweedie distribution.

```
from sklearn.linear_model import *
[...]
clf = TweedieRegressor()
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
```

Code 5.5: Tweedie-GLM implementation

TweedieRegressor is also inside the module linear_model of sklearn.

5.3.7 K-Nearest Neighbors

K Nearest Neighbors is a regression model that makes predictions based on the average of nearby data points.

```
from sklearn import neighbors
[...]
clf = neighbors.KNeighborsRegressor()
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
```

Code 5.6: K Nearest Neighbors implementation

KNeighborsRegressor is defined in the module **neighbors** of the sklearn library.

5.3.8 Decision Tree

The decision tree splits the dataset into subsets based on the most significant feature, the split points are chosen to minimize the variance for regression. The process is then repeated for each subset, creating a tree-like structure.

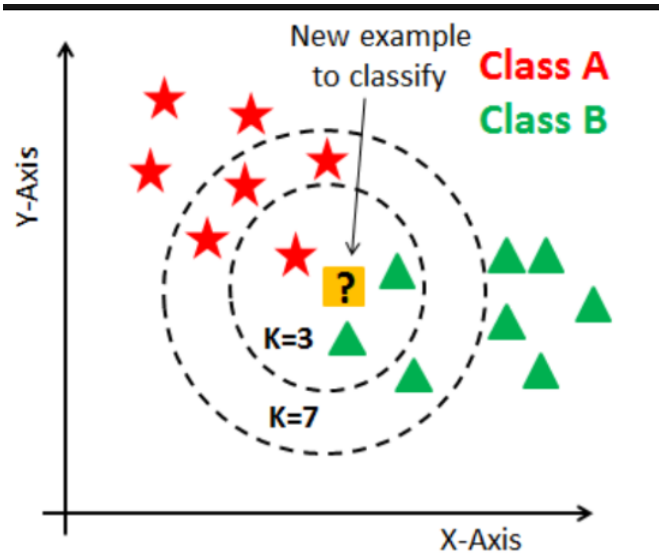


Figure 5.5: How KNN algorithm works, from [22]

To make predictions, the decision tree is traversed from the root to a leaf node, and the value associated with that leaf node is used as the final prediction. During the traversal, the algorithm evaluates the feature at each internal node based on the input features.

```
from sklearn import tree
[...]
clf = tree.DecisionTreeRegressor()
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
```

Code 5.7: Decision Tree implementation

DecisionTreeRegressor is defined in the module **tree** of the sklearn library.

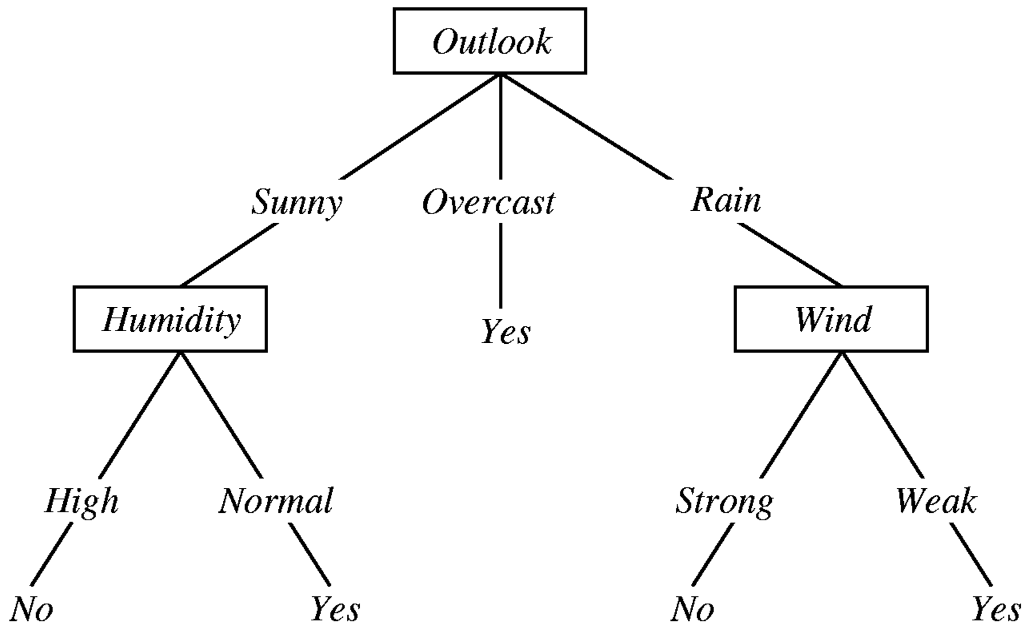


Figure 5.6: Example of a decision tree, from [20]

5.3.9 Extra Tree

The extra tree is a combination of multiple decision trees, extremely randomized, to improve the performances and robustness of decision trees.

ExtraTreeRegressor is defined in the module **tree** of the sklearn library.

```
from sklearn import tree
[...]
clf = tree.ExtraTreeRegressor()
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
```

Code 5.8: Extra Tree implementation

5.4 Evaluation of ML models' predictions

To evaluate the predictions of the ML models, we computed their ROC and Precision-Recall curves and compared them with the ROC and Precision-Recall curves of the software-tools AlphaFold-disorder and AlphaFold-disorder (SASA).

We will see the ROC curves and the Precision-Recall curves divided by ground truth used in the ML models training process.

5.4.1 Disorder PDB

In this pair of plots we see the ROC curves and Precision-Recall Curves, using disorder pdb as the ground truth. We can observe that AlphaFold-rsa (SASA) is a bit better than AlphaFold-rsa in both curves, a similar trend we have for AlphaFold-pLDDT (SASA) and AlphaFold-pLDDT. Finally the best ML model, LASSO, is within the average accuracy of the two software tools.

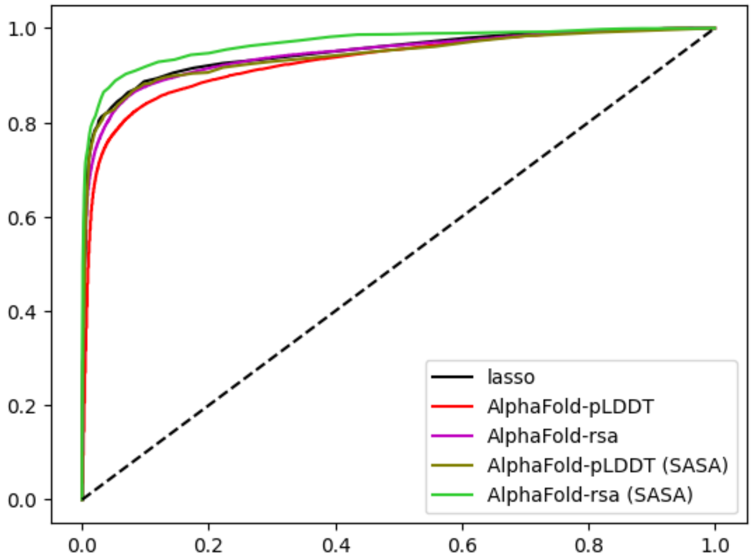


Figure 5.7: ROC curves - DisorderPDB ground truth

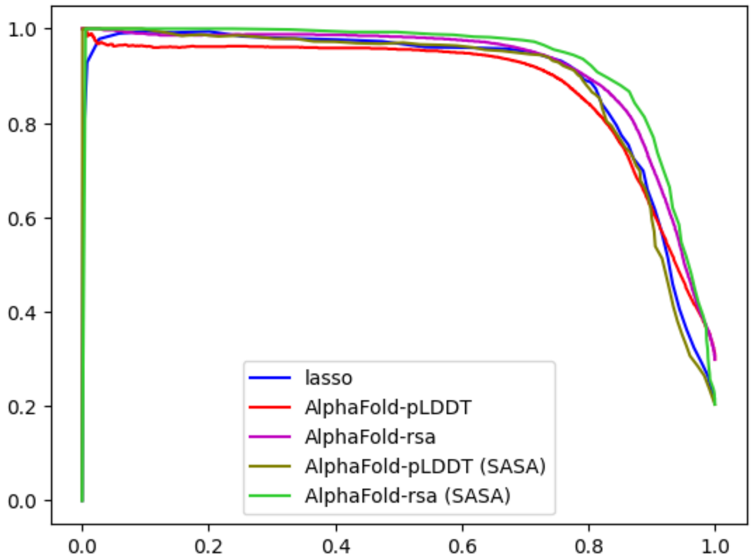


Figure 5.8: Precision-Recall curves - DisorderPDB ground truth

5.4.2 Disorder NOX

Here we used disorder NOX as the ground truth. Here AlphaFold-rsa (SASA) is quite better than AlphaFold-rsa in both curves, a similar trend we have for AlphaFold-pLDDT (SASA) and AlphaFold-pLDDT. Finally the ML model is within the accuracy of the two software tools, although it's better than both the predictors of the original AlphaFold-disorder.

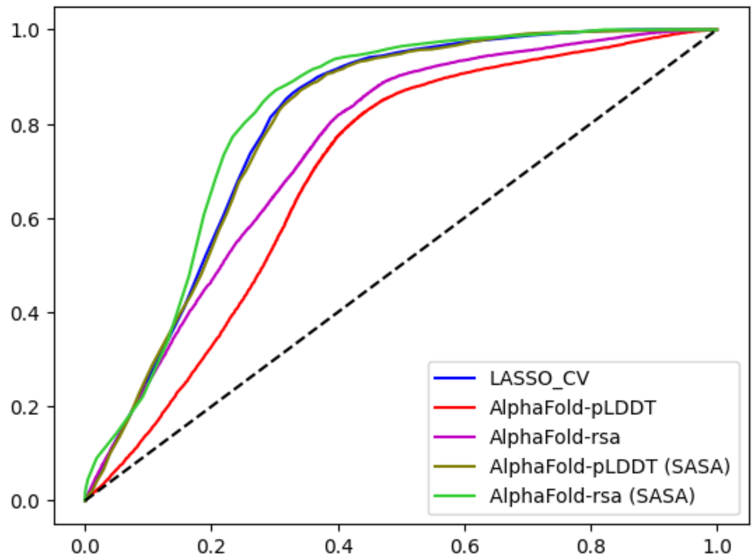


Figure 5.9: ROC curves - DisorderNOX ground truth

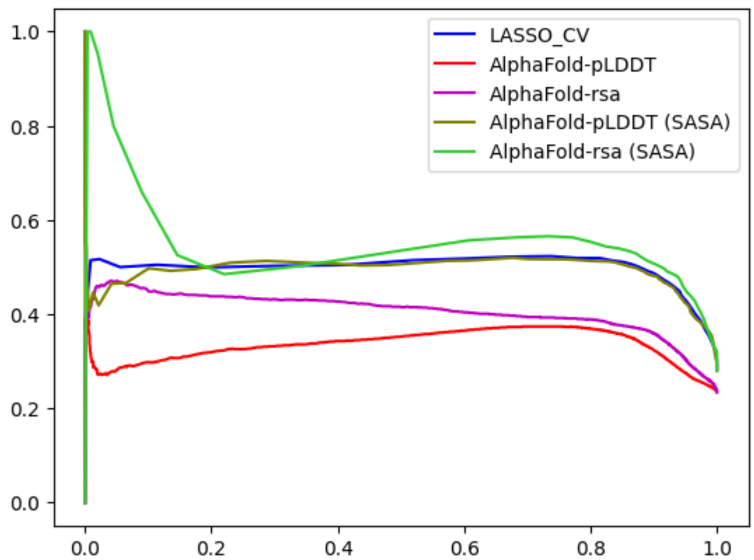


Figure 5.10: Precision-Recall curves - DisorderNOX ground truth

5.4.3 Binding

In the following figures, we consider binding as the ground truth. In both curves, for the major part, AlphaFold-binding (SASA) has better predictions than AlphaFold-binding and of the ML model.

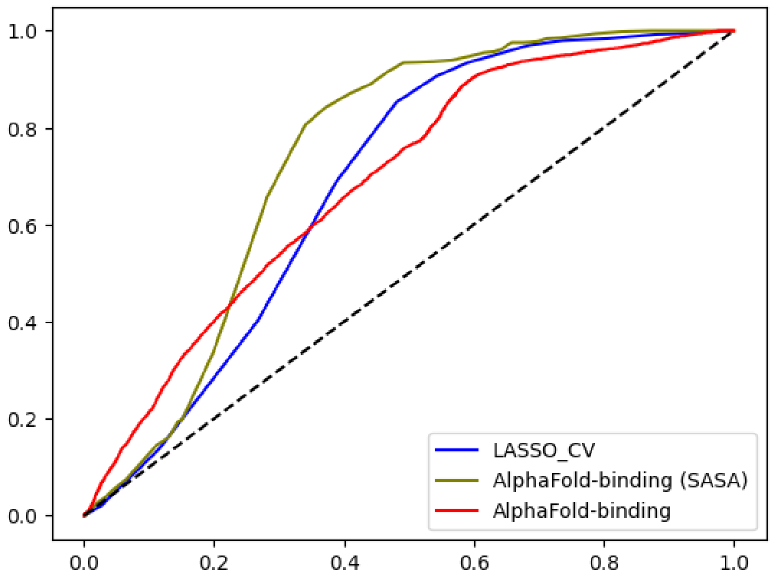


Figure 5.11: ROC curves - Binding ground truth

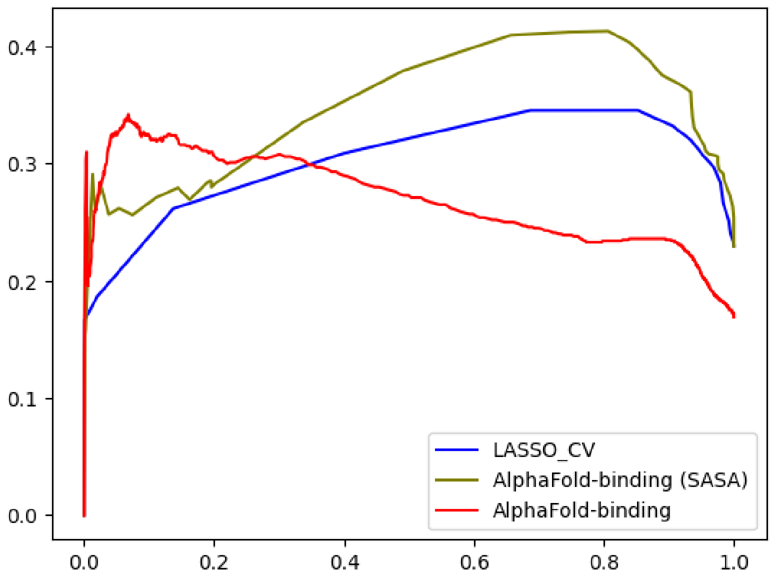


Figure 5.12: Precision-Recall curves - Binding ground truth

5.5 Helper Scripts for Analysis

5.5.1 Fetch proteins and functional annotations from DisProt database

First of all we downloaded the file .json from the DisProt website, in the download page. Then we used the library **json** of Python to extract from the json file, inside the json file we find all the information we want about DisProt proteins.

Here follows an overview of the structure of the JSON file: from the root of the json we have two keys, **data** and **size**. Size contains just the number of proteins in this json, and data contains an array of DisProt IDs. Each DisProt IDs have several key-value pairs, we are interested in **acc**, **disprot_id** and **regions**.

We are interested in the key **'acc'** of every DisProt protein because it is the UniProt ID: with these UniProt IDs we can fetch the proteins structure of all the proteins in DisProt with the AlphaFold database API.

```
import requests
[...]
list_absent = []
for i in uniprot_ids :
    url = 'https://alphafold.ebi.ac.uk/files/AF-'+i+'-F1-model_v4.pdb
    '
    r = requests.get(url, allow_redirects=True)
    if r.reason == 'OK' :
        filename = '' + i + '.pdb'
        open(filename,'wb').write(r.content)
    else :
        list_absent.append(i)
        print("Error "+str(r.status_code)+", "+r.reason+": "+i)
```

Code 5.9: Script to use AlphaFold database API

Then we are interested in the functional annotations of every DisProt protein, contained inside the key **'regions'**: an array of annotations for that protein.

5.5.2 Fetch ground truths arrays from CAID

We downloaded the ground truths data from the CAID web site, in the Data page. These ground truth files are in .FASTA, a file format for bioinformatic sequences.

In these files we have the ground truth for several DisProt ID, and for each of them to each residue is assigned a value between 0, 1 and '-'. '-' means no information, 1 is presence of disorder/binding and 0 is absence.

Finally we parsed these .FASTA files in pandas DataFrames and used them in our analysis. Down below an example of the first rows of the DataFrame for disorder-PDB ground truth.

	uniprot	disprot	amino_id	amino_name	disorder-pdb_GT
0	P06837	DP02342	1	M	1
1	P06837	DP02342	2	L	1
2	P06837	DP02342	3	C	1
3	P06837	DP02342	4	C	1
4	P06837	DP02342	5	M	1
5	P06837	DP02342	6	R	1

Table 5.2: DataFrame for Disorder-PDB ground truth

6

Conclusions

We developed a variation of the software-tool AlphaFold-disorder, replacing DSSP with PSEA and SASA. This was done mainly to avoid the dependency with DSSP and to explore an alternative way to obtain similar predictions, reproducing papers for this development as well.

AlphaFold-disorder (SASA) allows .fcz files, with the integrated FoldComp, so in the future we can store thousands of proteins with less storage required. A .fcz file contains the same data as a .pdb file with way lower size (around eight times smaller than .pdb).

Once the development was finished we started to evaluate the results, we started with explorative plots to gain some visual insight on the features. We used histogram plots to visualize the distributions of the features, related to the ground truth: both for each ground truth's class and then with both classes in the same plot for an easier visualization. Then we plotted scatterplots between pairs of features to visualize possible trends between two features, even in this case we did them for every ground truth, for both classes.

Then we tried to develop a ML model using AlphaFold-disorder (SASA) results to obtain predictions with higher accuracy. We used two thousands and five hundred proteins for training data, proteins from DisProt. The results were not as good as we hoped, but they just predicted what the software tool predicted, we probably a way bigger dataset for any meaningful increase in accuracy.

We compared AlphaFold-disorder (SASA) with AlphaFold-disorder with ROC curves and Precision-Recall curves. The variation we developed of the software tool has better ROC and PRecision-Recall curves than the original AlphaFold-disorder software tool. We can conclude the quality of results of the new software tool slightly improved or at least is on par.

References

[1] Annie Annie. *Amino Acid DPT.pptx*. 2023. URL: https://www.slideshare.net/AnnieAnnie18/amino-acid-dptpptx.

[2] Sarita Aggarwal Aradhana Bedi. *CK-12 CBSE Biology Class 12*. website. 6.1 Human Chromosomes and Genes. 2016.

[3] *CAID web application*. URL: https://caid.idpcentral.org/challenge.

[4] Omar Fitian Rashid. "Text Encryption Based on DNA Cryptography, RNA, and Amino Acid". In: Aug. 2021.

[5] Jaishree Gorane. *Write Short Note on Ribosomes*. https://classnotes123.com/write-short-note-on-ribosomes/. 2022.

[6] Joshua Hutchings and Giulia Zanetti. "Fine details in complex environments: the power of cryo-electron tomography". In: *Biochemical Society Transactions* 46 (June 2018), BST20170351. DOI: 10.1042/BST20170351.

[7] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (Aug. 2021), pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2. URL: https://doi.org/10.1038/s41586-021-03819-2.

[8] Wolfgang Kabsch and Christian Sander. "Dictionary of protein secondary structure: Pattern recognition of hydrogen-bonded and geometrical features". In: *Biopolymers* 22.12 (1983), pp. 2577–2637. DOI: https://doi.org/10.1002/bip.360221211. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/bip.360221211. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/bip.360221211.

[9] Andrey V. Kajava. "Tandem repeats in proteins: From sequence to structure". In: *Journal of Structural Biology* 179.3 (2012). Structural Bioinformatics, pp. 279–288. ISSN: 1047-8477. DOI: https://doi.org/10.1016/j.

jsb.2011.08.009. URL: https://www.sciencedirect.com/science/article/pii/S1047847711002371.

[10] Hyunbin Kim, Milot Mirdita, and Martin Steinegger. "Foldcomp: a library and format for compressing and indexing large protein structure sets". In: *Bioinformatics* 39.4 (Mar. 2023), btad153. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btad153. eprint: https://academic.oup.com/bioinformatics/article-pdf/39/4/btad153/49807920/btad153.pdf. URL: https://doi.org/10.1093/bioinformatics/btad153.

[11] G. Labesse et al. "P-SEA: a new efficient assignment of secondary structure from C trace of proteins". In: *Bioinformatics* 13.3 (June 1997), pp. 291–295. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/13.3.291. eprint: https://academic.oup.com/bioinformatics/article-pdf/13/3/291/1170655/13-3-291.pdf. URL: https://doi.org/10.1093/bioinformatics/13.3.291.

[12] Lukaves. *Illustration of the hierarchy of biological organization - from atom to the organism*. Illustration of the hierarchy of biological organization - from atom to the organism. URL: https://www.dreamstime.com/illustration-hierarchy-biological-organization-atom-to-organism-atom-to-organism-image212905226.

[13] Catherine A. Musselman and Tatiana G. Kutateladze. "Characterization of functional disordered regions within chromatin-associated proteins". In: *iScience* 24.2 (2021), p. 102070. ISSN: 2589-0042. DOI: https://doi.org/10.1016/j.isci.2021.102070. URL: https://www.sciencedirect.com/science/article/pii/S2589004221000389.

[14] Damiano Piovesan, Alexander Miguel Monzon, and Silvio C. E. Tosatto. "Intrinsic protein disorder and conditional folding in AlphaFoldDB". In: *Protein Science* 31.11 (2022), e4466. DOI: https://doi.org/10.1002/pro.4466. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/pro.4466. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/pro.4466.

[15] *Random Coil*. URL: https://d15.beauty/random-coil.

[16] Mahmood Rashid, Firas Khatib, and Abdul Sattar. "Protein preliminaries and structure prediction fundamentals for computer scientists". In: *http://arxiv.org* (Oct. 2015).

[17] Matheus Remigio. "Máquina de Vetores de Suporte SVM". In: (). URL: https://medium.com/@msremigio/m%C3%A1quinas-de-vetores-de-suporte-svm-77bb114d02fc.

[18] QCE Biology Revision. *Gene Expression*. website. 2020.

[19] Burkhard Rost and Chris Sander. "Conservation and prediction of solvent accessibility in protein families". In: *Proteins: Structure, Function, and Bioinformatics* 20.3 (1994), pp. 216–226. DOI: https://doi.org/10.1002/prot.340200303. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/prot.340200303. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/prot.340200303.

[20] Anshul Saini. *Decision Tree Algorithm A Complete Guide*. website. 2023.

[21] A. Shrake and J.A. Rupley. "Environment and exposure to solvent of protein atoms. Lysozyme and insulin". In: *Journal of Molecular Biology* 79.2 (1973), pp. 351–371. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(73)90011-9. URL: https://www.sciencedirect.com/science/article/pii/0022283673900119.

[22] Gattu Vijaya Kumar and Prasanta Sahoo. "A Recommendation System & Their Performance Metrics using several ML Algorithms". In: (June 2020).

Acknowledgments

I would like to express my deepest gratitude to my advisor Alexander Miguel Monzon and my co-supervisor Damiano Piovesan for their essential guidance and insight throughout this journey. I am also very thankful to the entire BioComputingUP laboratory for having welcomed me and provided help and support when I was in need.

I extend my gratitude to my family, my friends and my girlfriend Daniela for her love and support.