Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2017

Pedro Gabriel Fernandes Vieira

Deep learning para identificação de mutações genéticas patogénicas

Deep Learning for identification of pathogenic genetic mutations

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2017

Pedro Gabriel Fernandes Vieira

Deep learning para identificação de mutações genéticas patogénicas

Deep Learning for identification of pathogenic genetic mutations

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Sérgio Matos, Professor do Departamento de Electrónica e Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president	**Professora Doutora Ana Maria Perfeito Tomé** Professora Associada da Universidade de Aveiro
vogais / examiners committee	**Professor Doutor Joel Perdiz Arrais** Professor Auxiliar da Universidade de Coimbra
	Doutor Sérgio Miguel Aleixo de Matos Investigador Auxiliar da Universidade de Aveiro

agradecimentos / acknowledgements

É com enorme satisfação que aproveito para deixar um sincero agradecimento a todos os que me acompanharam durante estes cinco anos e que tornaram possível a realização desta dissertação, que culmina com o terminar desta importante etapa da minha vida.
Aos meus pais por me terem dado a oportunidade de poder frequentar o ensino superior e todos os sacrifícios que fizeram por mim.
A todos os amigos e restantes familiares pelo apoio incansável e por sempre acreditarem em mim na conclusão desta etapa.
Ao meu orientador, Doutor Sérgio Matos, pela disponibilidade, conselhos dados na realização desta dissertação e também por ter contribuído para desenvolver a minha autonomia, e pelo conhecimento transmitido.
A todos um sincero muito obrigado.

Palavras Chave Previsão de patogenicidade, Mutação genética, Machine Learning, Deep Learning, Codões, Ácido desoxirribonucleico

Resumo Nos dias de hoje, no qual o mundo está muito desenvolvido tecnologicamente, é possível efectuar várias tarefas em determinadas áreas que visam a ajudar a população mundial. Uma das áreas onde se tem investido mais em recursos tecnológicos é na área da bio informática. O crescimento desta área é sinal de melhoria na qualidade de vida da população, e essa melhoria pode passar por uma análise ao código genético, alertando-a de possíveis alterações genéticas ou até mesmo o aparecimento de doenças. Esta dissertação tem como objectivo efectuar uma análise ao código genético a fim de saber se uma alteração pode ser patogénica ou não. Numa primeira fase, são efectuados testes com classificadores clássicos, para saber qual o seu comportamento. De seguida, são efectuados novos testes mas desta vez usando modelos diferentes baseados em redes neuronais convolucionais para obter uma melhor previsão e resultados da mesma. Por fim, é feita a comparação entre cada um dos modelos adoptados para no futuro serem aplicados os respectivos modelos na área da bioinformática.

Keywords Pathogenicity prediction, Genetic mutation, Machine learning, Deep learning, Codons, Desoxyribonucleic acid

Abstract Nowadays, in which the world is highly developed technologically, is possible to perform several tasks in certain areas that aim to help the world's population. One of the areas where it has invested more in technological resources is in bioinformatics area. The growth in this area is a signal of improvement in quality of life of population, and this improvement may pass through an analysis of genetic code, alerting her of possible genetic changes or even the appearing of the diseases. This dissertation aims to perform an analysis to genetic code in order to know if an change may be pathogenic or not. In a first step, are performed tests with classical classifiers, to know their behaviour. Then, are performed new tests but this time using different models based on convolutional neural networks to get a better prediction and results of the same. Lastly, is done a comparison between each adopted classifier in order to be applied in the future the respective models in bioinformatics area.

Contents

List of Figures **iv**

List of Tables **v**

Acronyms **vii**

1 Introduction **1**

1.1 Motivation . . . 4

1.2 Objectives . . . 4

1.3 Contributions . . . 5

1.4 Organization . . . 5

2 State of the art **7**

2.1 Prediction tools and bias types . . . 8

2.1.1 Tools for pathogenicity prediction . . . 8

2.1.2 Two types of bias and datasets . . . 9

2.1.3 Tools results . . . 10

2.1.4 Conclusions . . . 12

2.2 PredictSNP2 Web platform . . . 14

2.2.1 Prediction tools, datasets and databases . . . 14

2.2.2 Performance Evaluation . . . 15

2.2.3 Conclusions . . . 16

3 Machine Learning **17**

3.1 Features extraction and classification . . . 19

3.2 Classical classifiers . . . 21

3.3 Deep Learning . . . 28

3.4 Evaluation . . . 40

3.4.1 Train and test . . . 42

3.4.2 Cross-validation . . . 43

4 Pathogenicity prediction **47**
4.1 Objectives 48
4.2 Technologies and libraries 49
4.2.1 Scikit-learn 49
4.2.2 TensorFlow 49
4.2.3 Keras 51
4.3 Datasets adopted 60
4.4 Predictions using classical classifiers 61
4.5 Predictions in nucleotide and codon sequence 65
4.5.1 One-Hot Deoxyribonucleic acid (DNA) sequence 65
4.5.2 65 One-Hot vector encode 68
4.5.3 One-Hot vector amino acid encode 70
4.5.4 Amino acid properties as a vector with ones and zeros 72
4.5.5 Venn diagram with amino acid properties 73
4.5.6 Amino acid physical-chemical properties 75
4.6 Models developed 76
4.6.1 Convolutional Neural Network 76
4.6.2 Complex Network 79

5 Discussion and Results **85**
5.1 Classical classifiers results 86
5.2 Nucleotide and codon sequence results 91

6 Conclusions and Future Work **97**

Appendices **104**

A Appendix **104**

List of Figures

1.1 Eukaryotic cell structure . . . 2
1.2 Prokaryotic cell structure . . . 2
1.3 DNA structure . . . 3
1.4 DNA sequence . . . 4

2.1 Exovar dataset . . . 10
2.2 Performance of prediction tools by ROC AUC . . . 11
2.3 Evaluation results of pathogenicity prediction tools . . . 12
2.4 Performance of nucleotide-based and protein-based prediction tools . . . 15

3.1 Structure of a simple neural network . . . 27
3.2 Workflow of artificial neural network . . . 29
3.3 Fully Connected Neural Network . . . 30
3.4 Workflow of convolutional neural network . . . 31
3.5 Mixed National Institute of Standards and Technology (MNIST) digit images . 34
3.6 ”Depth” changes of Neural Network (NN) architecture . . . 35
3.7 ”Width” changes of NN architecture . . . 35
3.8 Training time for Fully Connected Neural Network (FCNN) ”Depth” with Rectified Linear Unit (ReLU) function . . . 36
3.9 Training time for FCNN ”Width” with ReLU function . . . 36
3.10 Testing time for FCNN ”Depth” with ReLU nonlinearity function . . . 37
3.11 Testing time for FCNN ”Width” with ReLU nonlinearity function . . . 37
3.12 Accuracy classification for FCNN ”Depth” with ReLU function . . . 38
3.13 Classification accuracy for FCNN ”Width” with ReLU function . . . 38
3.14 Graphic representation of ROC_AUC metric . . . 41
3.15 Application of 10-fold cross-validation . . . 43

4.1 ReLU function and softplus . . . 53
4.2 Sigmoid function . . . 53
4.3 Tanh function . . . 54
4.4 MaxPooling layer operation . . . 57

4.5 Representation of geneID, mutation's position and mutation 60
4.6 Mutation performed from Guanine (G) to Adenine (A) 61
4.7 Mutation performed from Guanine (G) to Adenine (A) in codon sequence . . . 61
4.8 One-Hot Vector encode for each nucleotide 65
4.9 Mutation performed from Guanine to Adenine represented in vectors. 67
4.10 Example of reshape and vstack with the respective sequences. 67
4.11 65 One-Hot Vector encode for each codon. 68
4.12 Codons and amino acids 70
4.13 One-Hot vector encode for amino acids 71
4.14 Encoding with properties of amino acid 72
4.15 Venn's diagram for properties of amino acids. 73
4.16 Encoding according to Venn's diagram. 74
4.17 Amino acid physical-chemical properties 75
4.18 Convolutional Neural Network model 78
4.19 Complex Network model 79
4.20 Diagram of the Complex Network model 80

5.1 Average on dataset of Group k-fold score 87
5.2 Training and classification time for classical classifiers (average for 10 folds). Pre-processing time is between 2 and 11 seconds 89
5.3 Training and classification time for adopted encoding strategies (average for 10 folds). Pre-processing time is between 2 and 11 seconds 92
5.4 Accuracy score comparison between Convolutional Neural Network (CNN) and Complex Network model 93
5.5 Training and classification time for Complex Network model (average for 10 folds). Pre-processing time is between 2 and 11 seconds 94

A.1 Results for CNN and Complex Network model on Exovar dataset 110
A.2 Results for CNN and Complex Network model on Humvar dataset 110
A.3 Results for CNN and Complex Network model on PredictSNP dataset 111
A.4 Results for CNN and Complex Network model on Swissvar dataset 111
A.5 Results for CNN and Complex Network model on Varibench dataset 112

List of Tables

3.1 Framework complexity 39

4.1 Variants of each dataset 60
4.2 Properties of amino acids 72

5.1 Comparison of AUC with the State of the Art 88
5.2 Comparison of AUC between CNN and Complex Network models with the State of the Art 94

A.1 Exovar and Humvar results 104
A.2 PredictSNP and Swissvar results 104
A.3 Varibench results 105
A.4 Exovar and Humvar execution time 105
A.5 PredictSNP and Swissvar execution time 106
A.6 Varibench execution time 106
A.7 AUC score of Convolutional Neural Network 106
A.8 Exovar and Humvar DNA sequence execution time 107
A.9 PredictSNP and Swissvar DNA sequence execution time 107
A.10 Varibench DNA sequence execution time 107
A.11 AUC score of Complex Network 108
A.12 Exovar and Humvar execution time results for Complex Network 108
A.13 PredictSNP and Swissvar execution time results for Complex Network 108
A.14 Varibench execution time results for Complex Network 109

List of acronyms

ANN Artificial Neural Network

API Application Programming Interface

AUC-PR Area under the receiver operating characteristic curve Precision-Recall

AUC Area under the receiver operating characteristic curve

CADD Combined Annotation Dependent Depletion

CNN Convolutional Neural Network

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DANN Deleterious Annotation of Genetic Variants using Neural Networks

DNA Deoxyribonucleic acid

FatHMM Functional Analysis through Hidden Markov Models

FCNN Fully Connected Neural Network

FitCons Fitness Consequences of Functional Annotation

FN False Negative

FPR False Positive Rate

FP False Positive

GERP++ Genomic Evolutionary Rate Profiling

GPU Graphics Processing Unit

GWAVA Genome-Wide Annotation of Variants

HGMD Human Gene Mutation Database

LRT Likelihood Ratio Test

MAPP Multivariate Analysis of Protein Polymorphism

MASS MutationAssessor

MCC Matthews correlation coefficient

MLP Multi-layer Percepton

MNIST Mixed National Institute of Standards and Technology

MT2 MutationTaster-2

MV Majority Vote

NN Neural Network

PhD-SNP Predictor of human Deleterious Single Nucleotide Polymorphism

PP1 PolyPhen-1

PP2 PolyPhen-2

ReLU Rectified Linear Unit

ReLU Rectified Linear Unit

REST Representational State Transfer

RGB Red Green and Blue

RNA Ribonucleic acid

ROC Receiver Operating Characteristics

SIFT Sorting Intolerant From Tolerant

SNAP Screening for NonAcceptable Polymorphisms

SNP Single nucleotide polymorphism

SVM Support Vector Machine

Tanh Hyperbolic tangent

TN True Negative

TPR True Positive Rate

TP True Positive

Chapter 1

Introduction

Beforetime, people lived without any hygienic conditions compared to those of today. Throughout the ages, studies have been performed in all areas in order to improve the life's quality around the world. With these studies, began to emerge scientist, engineers, physicians, medicals, which help us to have better conditions to live. Some of these people contributed to the evolution of technology, which is applied increasingly in all areas. One of them is in bioinformatics.

In this area we can study the human's genetic code. It is important for us to know better our genetic code, mainly if it has some mutations that might cause some diseases, that is, if a mutation can be pathogenic. A mutation is a permanent change in DNA of a cell who can changes the function of the protein. Proteins are essential molecules for the body's behaviour. So, a mutation can change its structure and originate drastic impacts. DNA is located in cell's nucleus and carries the genetic information used for the growth, reproduction and development. In order to DNA make cells continuously work, there is the gene expression. Gene expression uses the genetic information located on DNA, transfers this information and converts it into protein. This is performed by transcription and translation processes. In the first process, DNA molecules copy their information by directing the synthesis of Ribonucleic acid (RNA) molecule [1] and in the second, RNA directs the synthesis of a protein. RNA is a molecule that plays biological roles in coding, decoding and expression of genes. The transfer of this information occur from DNA to RNA and then to Protein.

Relating to genetic mutations, they can be caused by an addition, subtraction or replacement of bases. They modify the genetic code and may originate a new sequence, which can modify the amino acid presented in the DNA. X-rays, substances present in the smoke, ultraviolet rays, dyes present in foods may be the main causes of these mutations [1].

To perform predictions on genetic code, it is essential to know some biological concepts.

As stated previously, genetic mutations occurs in cells. Cells are the basic and essential components of all living things. They may differ in functionality, or in appearance. There are *prokaryotic* and *eukaryotic* cells. Prokaryotic cells are very different from eukaryotic cells, because prokaryotic cells have a simple internal organization and these cells do not have nucleus. Eukaryotic cells have a more complicated internal organization including a defined membrane-limited nucleus [1]. Humans, animals and plants have eukaryotic cells because all of them have a nucleus. Below is shown the respective structures of eukaryotic and prokaryotic cells.

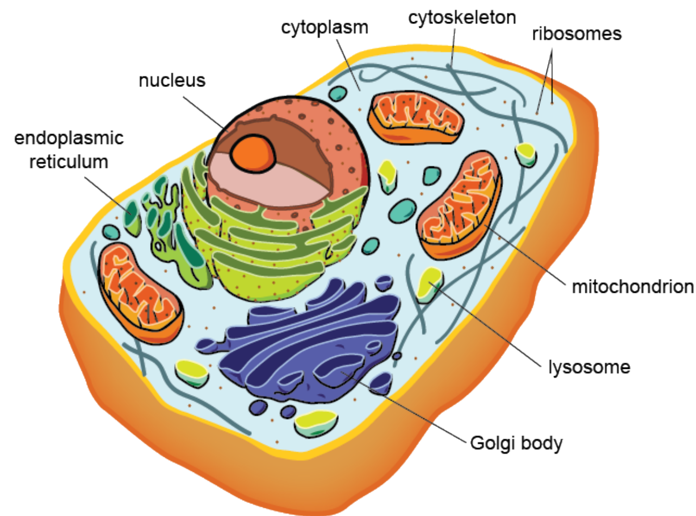


Figure 1.1: Eukaryotic cell structure [2]

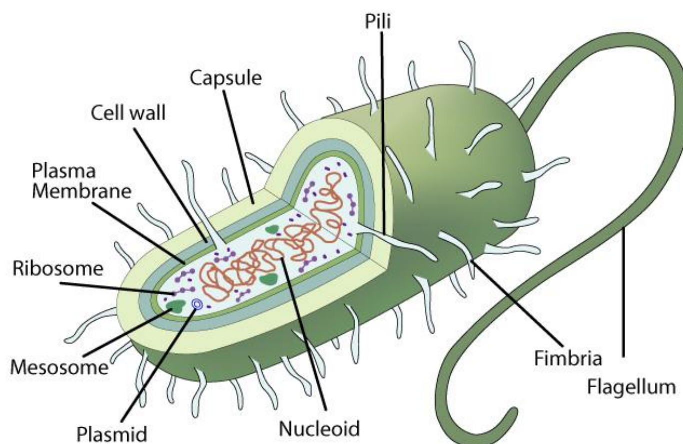


Figure 1.2: Prokaryotic cell structure [2]

The genetic data is encoded in the base sequence of the DNA, which has four bases that determines the sequence of amino acids encoded by the chain of *nucleotides* (organic molecules that serve as subunits, of nucleic acids like DNA and RNA), and these are composed by four organic basic types: *cytosine, adenine, guanine* and *thymine* [1]. A gene is the basic physical and functional unit of heredity. A gene is like a segment of DNA with a base sequence that encodes the amino acid sequence. In animals, genes are present on DNA molecules as *chromosomes* which are a set of nucleic acid sequences encoded as DNA within 23 chromosome pairs (Human Genome). The human's chromosomes of male and female are similar, but what distinguish them is the 23rd chromosome set. For males, the chromosome is XY and for females the chromosome is XX.

The genetic code describes how base sequences are converted into amino acid sequences during protein synthesis. The four organic basic types, previously mentioned, belong to the sequence of amino acids. The DNA sequence is divided into units of three bases, in which each set is a *codon*. A *codon* is a sequence of three DNA nucleotides that corresponds with an amino acid or stop signal. This means that the four bases in DNA can combine as total 4^3, i.e, 64 codons. The genetic code is composed by a full set of 61 codons, which 3 of them are stop signals [1].

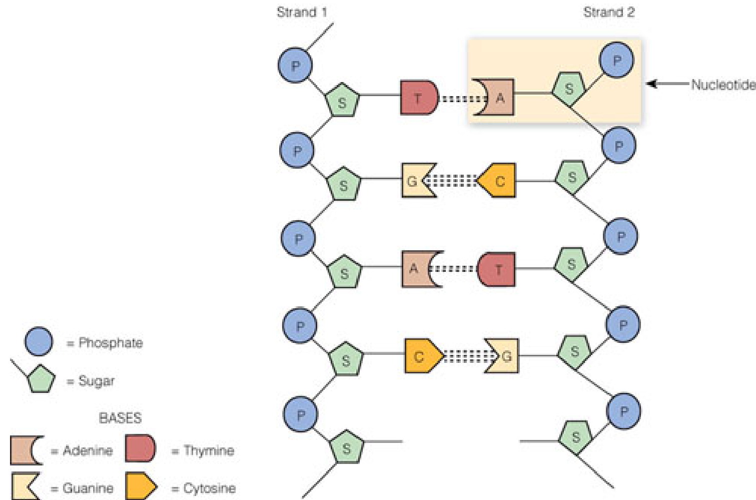


Figure 1.3: DNA structure [1]

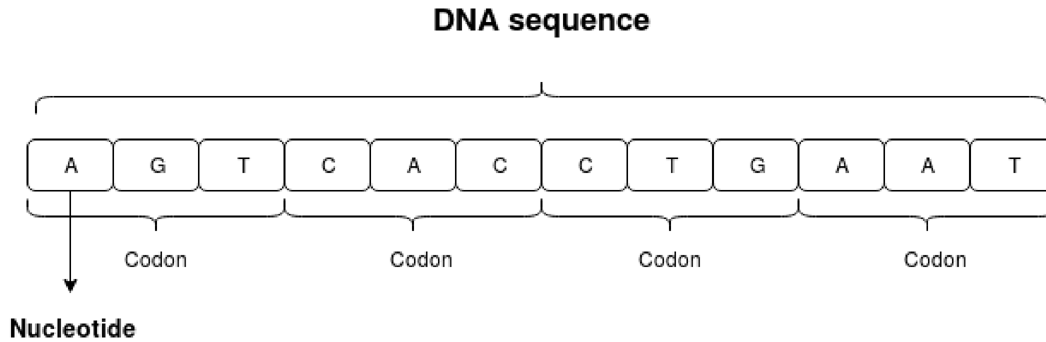


Figure 1.4: DNA sequence

1.1 Motivation

As mentioned in chapter 1, the evolution of technology allowed bioinformatics area to develop methods and software tools to work with biological data. With these all data and respective methods and tools, it is possible to perform an analysis on genetic code. This means an improvement in health and life's quality of a person, because these developed methods and software tools will help people know their genetic data and the respective consequences of a mutation occurred.

This is why this thematic is approached in this dissertation. A mutation in our genetic code can cause an impact for the rest of our life, and then may affect the next generations. Once these models developed, they can be integrated in a medical system to help doctors perform an analysis of person's genetic code, discover what disease may have, its origin, consequences, and advise or perform the right treatments in order to heal it.

1.2 Objectives

The main goal of this dissertation is to implement models based on *deep learning* that enables to predict if a certain mutation in genetic code will cause, or not, pathogenic effects. Pathogenic is the ability of an organism to contract a disease, and this may be hereditary. The probability to cause pathogenic effects performed by a genetic mutation may depend from other factors. The models will be trained and validated using the available datasets, and then compared with each other.

For training and validation of the models, will be considered different deep learning frameworks, including *Caffe* and *TensorFlow*. Caffe has an extensible code, that promotes active development, good speed for research experiments and an expressive architecture that encourages application and innovation. TensorFlow is an open source deep learning framework that uses numerical computation and data flow graphs.

In this dissertation it's also performed an overview of prediction tools used in other works,

in order to evaluate the performance of these tools in different datasets. It's also performed an analysis of the performance of each model, in order to know which is the most efficient. This will contribute to know which are the best tools to perform predictions in a genetic sequence.

1.3 Contributions

Along with this dissertation, there is an article named "Analysis of classifiers for identification of pathogenic mutations" which describes the methods used in this dissertation to perform pathogenicity predictions. This article was submitted to INForum.

1.4 Organization

This dissertation is splitted into 6 chapters. The remaining chapters are organized as follows:

- **chapter 2**: presentation of the state of the art. In this chapter are presented some concepts about genetic mutations, prediction tools and its evaluation, pathogenicity prediction, datasets for prediction and a comparison about the results obtained from some studies.
- **chapter 3**: introduction to machine learning and deep learning and the respective frameworks. Some concepts of features extraction from genetic code and comparison between deep learning frameworks.
- **chapter 4**: description of the each method developed to implement the solution. In this chapter there's an explanation of every step performed in the implementation of the solution.
- **chapter 5**: in this chapter is presented the analysis of results obtained from the implemented solution, and a description of each test methodology.
- **chapter 6**: final conclusions about the chosen methods for the implementation of the solution, and obtained results of this work, also referring potential improvements or other solutions for future work in this area.

Chapter 2

State of the art

In a world of constant technological growing, the resources to analyze cells, chromosomes, body's composition, DNA, genetic code and prediction of mutations, are rather extensive and more accurate.

These resources may be some prediction tools, for example classifiers, which are described in this chapter and the respective results applying each tool on the mutated genetic code. These descriptions are based on the works [3] and [4]. The work [3] tells about two problems detected when performing evaluation of some tools used to predict the impact of these mutations, and the work [4] is based on a software prediction tool for predicting non-synonymous genetic variants, that is, nucleotide mutations that may alter amino acid sequence of a protein, and consequently, cause human diseases.

This chapter also mentions an unified platform, named PredictSNP2, for accurately evaluating Single nucleotide polymorphism (SNP) effects, i.e., impacts on the simplest form of DNA variation among all individuals. This is performed by exploiting the different characteristics of variants in distinct genomic regions, based on work [5].

2.1 Prediction tools and bias types

2.1.1 Tools for pathogenicity prediction

Missense mutation is a point mutation, which changes one DNA nucleotide that causes the substitution of a different amino acid in a protein. This may cause a pathogenic mutation.

According to what was introduced in chapter 1, the genetic mutations are the focus of a large number of studies. One of the aspects studied in these studies is the impact of mutations, mainly the classification of pathogenicity. In this area, there are many studies which developed many tools to perform prediction on the genetic code and improve the respective classification. Some of these tools are PolyPhen-2 (PP2) [6], Sorting Intolerant From Tolerant (SIFT) [7], Functional Analysis through Hidden Markov Models (FatHMM), MutationTaster-2 (MT2) [8], MutationAssessor (MASS) [9], Combined Annotation Dependent Depletion (CADD) [10], Likelihood Ratio Test (LRT) [11] and Genomic Evolutionary Rate Profiling (GERP++) [12]. These tools are used to predict possible impacts on an amino acid substitution on organism structure. In the article [3] was demonstrated that in a study of ten tools on five datasets, a comparative evaluation of these tools is hindered by two types of bias (in the article they mentioned as ”circularity” types). The origin of these types of bias is due to the variants from a protein occurring both in the datasets used for training and for evaluation of these tools [3]. With these types of bias, the results become more optimistic than without them.

In addition to these tools, there are others already available in order to perform also pathogenicity prediction. These tools are the FatHMM weighted (FatHMM-W) and unweighted (FatHMM-U) [13], and phyloP [14]. These tools are used to separate pathogenic variants from neutral variants (neutral or non-pathogenic variants). The usage of this tools is important because facilitates the prediction. Phylop and SIFT are used to measure sequence conservation, PP2 is used to evaluate the impact of amino acid substitution on protein structure or function. CADD is used to quantify the pathogenic potential of a variant, based on diverse types of genomic information [3]. MT2 is a free web application to evaluate DNA sequence variants which may cause a disease. MASS is based on a prediction of functional impacts of amino acid substitutions in a protein, for example, mutations discovered in a cancer or some other disease. LRT is based on a genomic dataset of some vertebrate species and applies a likelihood ratio test to compare the null model that each amino acid is evolving to the alternative model of evolution. GERP uses maximum likelihood evolutionary rate to produce position estimators and GERP++ is a programming approach to define constrained elements.

Once there are different methods that can perform pathogenicity prediction, it is essential to know if one or several tools outperform all others in prediction accuracy. Here, we talk about the two problems when performing a comparative evaluation of these tools, mentioned in the previous section. They are called as the two types of bias.

2.1.2 Two types of bias and datasets

The term bias, in the article [3], is used to describe that predictors are evaluated on variants that were used to train prediction models. The first type of bias is related to the overlaps between datasets that were used to train and evaluate the models. Tools such as MT2 [8], PP2 [6], MASS [9] and CADD [10] require a training dataset to determine the features of the model. These tools may take the risk of capture very own characteristics of their training set and may become not generalized, i.e., the training set gets biased when applied on a new data. To avoid this overfitting problem, the tools must be evaluated on variants that were not used for the training of these tools.

The second type of bias is based on a property present on databases. Variants from the same gene are classified as pathogenic or neutral. A classifier that predicts pathogenicity based on known information about certain variants in the same gene gets better results. These types of bias may be originated by the combination of these tools mentioned [3].

All of these tools mentioned require datasets which are composed by DNA sequences. In all datasets, there are some variants that are not possible to determine nucleotide and amino acid changes. So, they must be excluded, due to the fact that they could interfere in the final results and execution time. This means that before performing predictions, the data must be pre-processed, i.e., might be necessary to remove, insert, or even replace some variants. Datasets can be different among each other. They can have different number of samples, features and even its content, i.e., we can have a dataset which contains the values of each feature that were calculated by mathematical formulas, or in the case of this dissertation, the genetic sequence represented by the nucleotides.

There are some datasets used frequently when performing pathogenicity prediction. These datasets are also used in this dissertation to perform the prediction of pathogenicity. They are the *HumVar* [6], *ExoVar* [15], *VariBench*, *predictSNP* and *SwissVar* dataset. The non-overlapping variants of Varibench dataset might originate *VariBenchSelected* dataset. PredictSNP dataset was used to exclude all variants that overlap with HumVar, ExoVar and VariBench and the resulting dataset was called *predictSNPSelected*. Swissvar may originate *SwissVarSelected*, which consists into exclude all variants overlapping with the other datasets (HumVar, ExoVar, VariBench and PredictSNP). This dataset should be the one that contains the newest variants across all others datasets. The biggest dataset of this five is the Humvar. In this dissertation, Humvar dataset contains more than 40 thousand sample, thus, this dataset will be the one that takes more time to pre-process the data, train and classify a model. An example of a dataset's content is shown below in Figure 2.1:

```
NM_000016.4:c.577A>G,1,6.045005314036012,0.989913166,-0.000438809,
NM_000016.4:c.985A>G,1,6.045005314036012,0.989913166,0.000012994,0
NM_000017.2:c.1058C>T,1,6.023447592961033,1.227456212,0.000765443,
NM_000017.2:c.1138C>T,1,6.023447592961033,1.227456212,0.002587199,
NM_000017.2:c.575C>T,1,6.023447592961033,1.227456212,0.002575874,0
NM_000017.2:c.973C>T,1,6.023447592961033,1.227456212,0.002587199,0
NM_000018.3:c.1600G>A,-1,6.486160788944089,1.184138656,0.000008345
NM_000019.3:c.1138G>A,1,6.059123195581797,1.036389589,0.000087738,
NM_000019.3:c.472A>G,1,6.059123195581797,1.036389589,0.000000358,0
NM_000019.3:c.547G>C,1,6.059123195581797,1.036389589,-0.000162244,
NM_000019.3:c.901G>C,1,6.059123195581797,1.036389589,-0.000059485,
NM_000019.3:c.997G>C,1,6.059123195581797,1.036389589,-0.000391841,
NM_000019.3:c.997G>C,1,6.059123195581797,1.036389589,-0.000391841,
NM_000020.2:c.1031G>A,1,6.222576268071369,1.245653868,-0.000015020
NM_000020.2:c.1121G>A,1,6.222576268071369,1.245653868,-0.000736117
NM_000020.2:c.1133C>T,1,6.222576268071369,1.245653868,0.002699018,
NM_000020.2:c.1232G>A,1,6.222576268071369,1.245653868,-0.000736356
NM_000020.2:c.1460A>C,1,6.222576268071369,1.245653868,0.001428485,
```

Figure 2.1: Exovar dataset

In dataset present in Figure 2.1, the characters before the first commas, represents the gene's identifier, position where a mutation occurs and the respective substitution of nucleotides. Then, between the first and the second commas is the class of variant, in which 1 means that is pathogenic and -1 is neutral. After these second commas until the last column are the values of the data to be used to train and classify the tools.

2.1.3 Tools results

These tools were tested on all datasets and the results were discussed and compared. Has been performed a comparison between the results obtained for each selected datasets, present in the Figure 2.2. Then is performed the same but comparing the results of the tools on each dataset present in Figure 2.3.

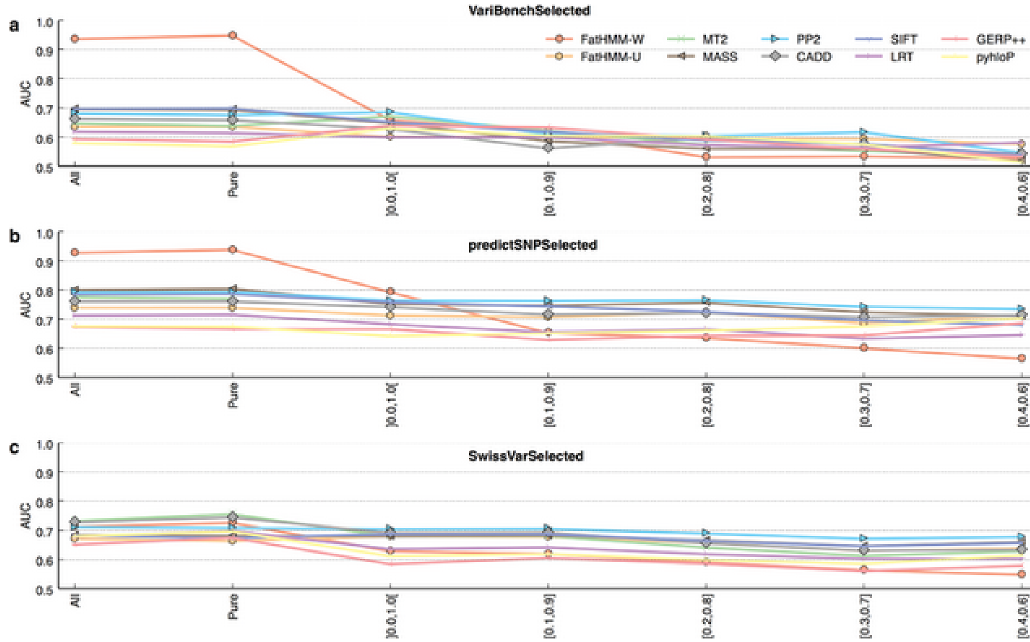


Figure 2.2: Performance of prediction tools by AUC metric. [3]

The results displayed above in Figure 2.2 show that the tool FatHMM-W performs well on pure proteins (proteins with only one class of variant) for VariBenchSelected and PredictSNPSelected but not on the mixed proteins (proteins with pathogenic and neutral variants), due to its weighting scheme, but this tool is outperformed by the other tools on balanced proteins (50% of neutral and 50% of pathogenic variants). PP2 outperforms other tools in mixed categories for PredictSNPSelected and SwissVarSelected datasets [3]. The results are measured by the scoring metric Area under the receiver operating characteristic curve (AUC) which will be explained in chapter 3. The evaluation of the weighting scheme may generate biased subset, i.e., a subset that contains only variants of the same kind, but this can be solved by applying cross-validation, which will be explained in detail in chapter 3.

Then, as mentioned previously, are obtained the results for all of these tools, but performing a comparison on each of the five datasets.

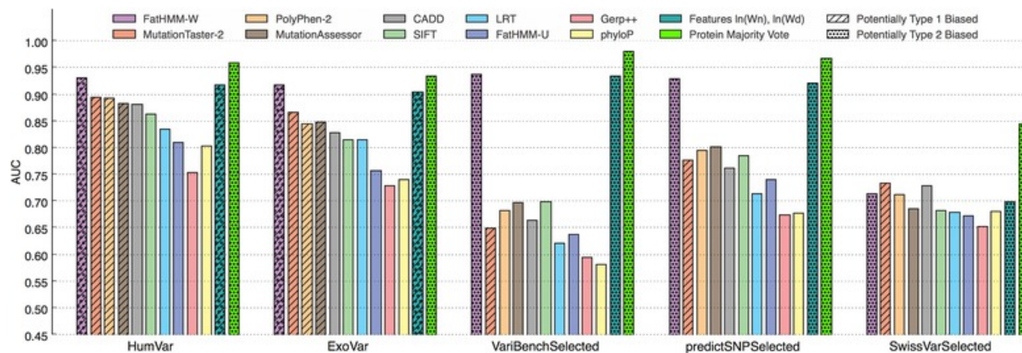


Figure 2.3: Evaluation results of pathogenicity prediction tools. [3]

Hatched bars in Figure 2.3 indicate that the evaluation data is to train the corresponding tool, and these results may suffer from overfitting. The dotted bars indicate that the tools are biased, because of type 2 of bias [3]. In the graphic above, on HumVar and ExoVar, the four best performing methods were trained on these datasets. Across the first four datasets, FatHMM-W outperforms all other tools. Then, MT2 performs well on Humvar and Exovar but bad on VariBenchSelected and SwissVarSelected. The same happens for PP2 which performs better than MT2 on VariBenchSelected and predictSNPSelected. MASS performs better than these two tools on VariBenchSelected and predictSNPSelected, but worse on HumVar and SwissVarSelected.

2.1.4 Conclusions

As we can see, has been performed a comparative evaluation of pathogenicity prediction tools and demonstrated that there are two types of bias which harm comparison of pathogenicity prediction tools. This was performed in order to know if there are differences in quality of pathogenic prediction tools when evaluated on a large number of variant databases [3]. The discovery of novel risk genes may become difficult if these two types of bias were ignored.

Type 1 bias is studied in Machine learning (explained in chapter 3), and is originated from an overlap between training and evaluation datasets, performing worse on different variants. The creation of *Selected* datasets avoids this type of bias. In type 2 bias, predicting the pathogenicity of a variant based on the pathogenicity of all other variants may lead to better results, but will fail to classify variants in proteins that contain neutral and pathogenic variants. So, it is possible to conclude that is due to this type bias that the AUC results of FatHMM-W on four out of five datasets are excellent.

The Figure 2.3 shows that the remaining tools get a better performance in Humvar and Exovar due to its weighting scheme. The AUC metric results for Majority Vote (MV) with type 2 bias are always the best for each dataset, because for each of the five evaluation datasets, the data is splitted into 10 subsets, which nine of them are used as training data and one

as test data. Then, in each iteration, another subset is determined as the test data and the remaining are the training data (cross-validation). MT2, PP2 and MASS are one of the best predictions tools, mainly when applied on Humvar and Exovar dataset, achieving an evaluation nearly of 90%. Although these values are lower in *Selected* datasets than in Humvar and Exovar, they are still good prediction tools.

2.2 PredictSNP2 Web platform

PredictSNP2 is a web platform developed for evaluation of the SNP effects and to perform predictions on some tools such as Deleterious Annotation of Genetic Variants using Neural Networks (DANN), FunSeq2 [16], Genome-Wide Annotation of Variants (GWAVA), PredictSNP2, Multivariate Analysis of Protein Polymorphism (MAPP) [17], Predictor of human Deleterious Single Nucleotide Polymorphism (PhD-SNP) [18], PolyPhen-1 (PP1) [19], Screening for NonAcceptable Polymorphisms (SNAP) [20], predictSNP1 [21], CADD [10], FatHMM and Fitness Consequences of Functional Annotation (FitCons) (*Fitness Consequences of Functional Annotation*). SNP is a variation in a nucleotide at a specific position in the genome. PredictSNP2 is a natural extension of previously published PredictSNP1 [21] tool. PredictSNP1 it is used to analyse substitutions in an amino acid sequence. PredictSNP2 completes PredictSNP1 by evaluating the effects of nucleotide variants, stored in any region of the genome [5]. This web platform may be like a continuation of what was previously mentioned relating to the results of prediction tools and is available at *http://loschmidt.chemi.muni.cz/predictsnp2*.

2.2.1 Prediction tools, datasets and databases

Obviously that to perform predictions is necessary to use datasets. The data on the datasets for the web platform, was splitted into five subsets by classifying the variants according to their consequences determined by ANNOVAR [22] software [5].

The prediction tools mentioned previously were used for evaluation, optimization and integration into the PredictSNP2 web portal. CADD is a tool for scoring the deleteriousness of single nucleotide variants. Its predictions are based on a logistic regression model. Also, it is trained on a constructed dataset of mutations obtained from differences between the human and chimpanzee genome [5]. DANN is a deep neural network-based classifier that enables capture non-linear relationships among features. FatHMM-MKL [23] uses an Support Vector Machine (SVM) model to evaluate the functional impact of variants, which was trained on Human Gene Mutation Database (HGMD) [23]. GWAVA [24] tool is based on a random forest classifier, which is used for the analysis of regulatory variants [5]. FunSeq2 uses a scoring system to assess the impact of variants. FitCons estimates functional impact of variants, using fingerprints [5]. The databases used on the web platform are the ClinVar [25], dbSNP [26], Ensembl Genome Browser [27], GenBank [28], HaploReg [29], OMIM [30], RegulomeDB [31] and UCSC Genome Browser [32].

2.2.2 Performance Evaluation

The performance evaluation of these tools is done using statistical metrics used and mentioned previously in the evaluation of performance with the two types of bias already approached in this dissertation (*specificity, accuracy, precision, recall/sensivity* will be explained further ahead). This evaluation is performed on the datasets mentioned previously (such as OMIM, GenBank, RegulomeDB, etc). The performance of selected nucleotide based prediction tools were compared with some protein-level tools. The protein-level tools chosen to perform the comparison were MAPP, PhD-SNP, PP1, PP2, SIFT, SNAP and PredictSNP1. To perform the comparison was used ANNOVAR [22] software to convert nucleotide variants into amino acid format.

The results of the respective performance comparison are shown in the following image:

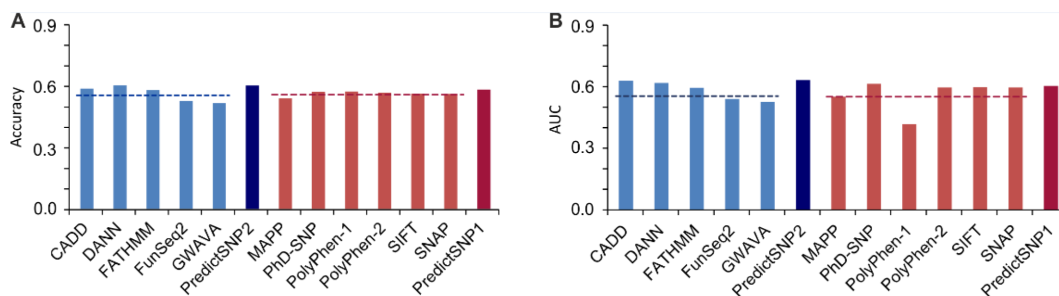


Figure 2.4: Performance of nucleotide-based and protein-based prediction tools. [5]

Blue bars represent the values for nucleotide-based tools, and the red bars represents the values for protein-based tools. The graphic A shows the accuracy obtained for each tool, and the graphic B shows the AUC for each tool. The horizontal dashed lines represent average performance for each tool.

For nucleotide-based prediction tools, FatHMM and predictSNP2 are those that have the best Accuracy due, as already said previously, its weighting scheme (nearly 80%). CADD, DANN and FatHMM are the nucleotide-based prediction tools that have the best accuracy of all. Relatively to FunSeq2 and GWAVA, these are not good nucleotide-based prediction tools because its accuracy is below average performance values. As we can see in Figure 2.4, this analysis can be performed also for AUC. For protein-based prediction, PhD-SNP, SIFT and PredictSNP1 have an accuracy higher than 70% and its accuracy values are bigger than the average performance values. For AUC, happens the same situation, which PhD-SNP, SIFT and PredictSNP1 have its accuracy values bigger than the average performance values. From all of this, we can conclude that predictSN1 and predictSNP2 are the best protein-based and nucleotide-based prediction tools, respectively.

2.2.3 Conclusions

Taking into account the tools mentioned previously, some of them could be also considered to be use by this web platform, because they achieved great results on pathogenicity prediction. Besides that, these results obtained from tools used in this web platform are not so good, because even that some of them are above of the average, the respective scores of each metric are below of 0.7. The fact that these results are not so good may be also due to the datasets used by each tool. In the study [3] the tools are trained and classified on Exovar, Humvar, predictSNP, Varibench and Swissvar. This web platform uses some tools that are also used in the study [3], thus this could be considered the usage of these datasets on the web platform. This results will be later compared to those obtained in this dissertation in order to know if the models that will be created, trained and classified are better than some of these studied.

CHAPTER 3

Machine Learning

Machine Learning is a method of data analysis, which automates the building of the models using algorithms to learn from the data. Machine learning emerged from studies about pattern recognition and theory of computational learning in artificial intelligence, in which were the first areas where this are was applied. Machine Learning has experienced a great growth and it is commonly used in data mining, natural language processing, image recognition, and expert system. Nowadays, this area is being used by Google autonomous cars, Amazon and NetFlix use it on online recommendations, fraud detection, comments on Twitter and in many other applications. One of the reasons that Machine learning has being grown is because exists more data variety and quantity available. Other is the computational processing, which is more powerful, and the storage of the data in an accessible way. All of these facts make possible to develop models able to analyse more quantity and complex data, thus provide faster and precise results.

Machine Learning workflow can be broken down into 4 steps: *pre-processing, feature extraction, training* and *testing*. First step (pre-processing) aims to transform the info into a dataset, in order to step two (feature extraction) perform an extraction of the relevant features to training the model (training step). After the model is well trained, we can perform the last step, which is testing the model. There are four important types of Machine learning: *Supervised Machine learning, Unsupervised Machine learning, Semi-supervised Machine learning*, and *Reinforcement Machine learning*. Each one is described in detail in the following headings.

Supervised Machine learning

Supervised Machine learning is the most used type by Machine learning models and classifiers. In this type, we have input variables (x) and an output variable (y) and it is used

an algorithm to learn the mapping function from the input to the output like:

$$y = f(x)$$

The main goal of this type of Machine learning is to get closer the mapping functions so well that when we have new input data (x), we can predict the output variables (y) for that data. The process of an algorithm learning has a supervisor, in order to assure and supervise the learning process. The algorithm makes predictions on the training data, which is corrected by the supervisor, until it reaches the optimal level performance. The supervised learning problems can be aggregated into two problems: *classification* and *regression*. Classification problem is when the output variable represents a category, such as "disease" and "no disease" (which is the focus of this dissertation). Regression problem is when the output variable represents a real value, such as "dollars" or "height". Some examples of popular Supervised Machine learning algorithms are:

- **Linear regression**, for classification and regression problems
- **Random forest**, for classification and regression problems
- **Support vector machines**, for classification problems

Unsupervised Machine learning

Unsupervised learning is where we only have input data (x) and no corresponding output variables, i.e., there is no labeled data, but only unlabeled. There are no training or tested examples used in this process. The main goal of this type of Machine learning is to discover some similar groups within the data (clustering).

Unlike Supervised Machine learning, this type of Machine learning has no correct answers and there is no teacher. Algorithms and models must discover and present the interesting structure in the data. Unsupervised learning problems can be aggregated into *clustering* and *association* problems. Clustering problem is where it is pretended to discover the inherited groupings in the data, i.e., it is like grouping customers by purchasing behavior. Association problem is where it is pretended to discover rules that describe large amounts of the data, such as people that perform X also tend to perform Y. Some examples of popular Unsupervised Machine learning algorithms are:

- **k-means**, for clustering problems
- **Apriori algorithm**, for association rule learning problems

Semi-supervised Machine learning

Semi-supervised Machine learning is a class of Supervised Machine learning, which uses both labeled and unlabeled data for training. This Machine learning type can be considered as the center of Supervised Machine learning, which uses completely labeled training data, and Unsupervised Machine learning, which not contains any labeled training data.

It is due to the fact that this type of Machine learning uses unlabeled data that enables to improve learning accuracy, because the unlabeled data does not have any cost to be generated. This means that labeled data require human skills in some areas, such as image processing, audio transcription, etc, and this requires cost of the resources used in data labeling. This type of Machine learning works better when using small amounts of labeled data and a large amount of unlabeled data.

The main goal of Semi-supervised Machine learning is to understand how a combination of labeled and unlabeled data may change the learning behaviour. Then, according to this, is possible to design the models that take advantage of the combination. This type of Machine learning can be in human category learning, because almost all inputs are unlabeled.

Reinforcement Machine learning

Reinforcement Machine learning that bases in behaviour psychology. This type of Machine learning concerns with how software agents take decisions in an environment. This problem is studied in games theory, control theory, operations research, genetic algorithms, etc. The environment in this Machine learning problem is formulated as a Markov decision process, in which for this context uses dynamic programming techniques. Also, this type of Machine learning allows to determine automatically the behaviour of the machine or software agent based on feedback from the environment, in order to maximize its performance.

Unlike classical techniques, Reinforcement Machine learning does not need knowledge about the Markov decision processes. This Machine learning problem differs from Supervised Machine learning problem in that input/output pairs are never presented. Furthermore, on reinforcement learning exists more focus on performance, which involves a balance between exploration and exploitation.

3.1 Features extraction and classification

As we stated previously, it is necessary to create algorithms or predictive models in order to predict if a mutation in genetic code is pathogenic. This means that is essential to have a dataset only with relevant data, which is possible with feature extraction. A feature is a measurable property of an animal, human, object being observed. For example, if we are trying to predict the type of a pet, the input features might be home region, family, age, animal's breed, etc. Its labels (outputs) might be dog, cat, horse, etc. Feature extraction is

the process of automatically extract the features in the data that are most useful or relevant to the predictive modeling problem.

In the context of this dissertation, can be considered some features which are present in the study [33]. This study relates that a total of 152 features (variant, gene, and protein characteristics) were extracted from 55 thousand variants. For gene features, several characteristics are calculated considering the *codon usage, codon context, stop codon, nucleotide repeats, variant position in gene and codon, GC-content* (guanine-cytosine content), *proximity to splicing sites, variations of nucleotide, etc.* Relative Synonymous Codon usages and codon pair scores for all codon pairs are calculated using human genome. From human genome, are also considered some features, such as length of chromosome, its base pairs, variations, etc. Stop codons were calculated with TGA, TAG and TAA codons present in all possible codons. Nucleotide repeats were calculated as the number of consecutive repeated nucleotides in a sequence of seven codons. The proximity of splicing sites is performed by the retrieving of exon positions for the affected transcript, in order to get the distance to the nearest splicing site. The presence of exon splicing regulatory sequences were obtained using two lists of exon splicing enhancers and silencers [34]. This is done in order to get a score that represents the amount matches between the affected region and a splicing regulation sequence, and this score is obtained by measuring the difference between the affected region and every splicing regulatory sequence on the lists [33].

For protein features, are taken into account some changes in chemical properties of amino acid. These chemical properties are *hydropathy, polarity, mass, volume, acidity, net charge*, and *isoelectric point* (these properties will be used further ahead in chapter 4). Also, are taken into account the evaluations of the changes performed in the affected region, and structural changes by predicting the protein secondary structure [35].

Feature extraction gives the ability of choosing features that will give good or better accuracy requiring less data. It can improve the accuracy and reduce the execution time by using extraction methods to identify and removed irrelevant and redundant attributes from the data. These redundant data decreases the accuracy of a predictive model. A good predictive model must have a simple complexity, i.e., simple to explain, understand and use. This is why it is important to perform feature extraction. Feature selection is also used in some works when it is pretended to measure the results only using a certain features. After features extraction process, it is essential to perform the respective classification of these features, in order to predict if a mutation on these genetic features is pathogenic or neutral. To achieve this, is needed to build a classification model to perform the respective predictions. To perform this prediction, was used Machine learning classifiers to know the one that better matches prediction needs. Classifiers such as *Decision Trees, Linear regression, Naive Bayes, NN, Logistic regression, SVM*, etc, will be used and explained in detail in this chapter further ahead.

In the study [33], the data was divided into 5 parts: 4 for training and 1 for testing. In the final tests, the data was divided into 20 parts, in order to improve the assessment of predictive power. To avoid loss of information between instances of dataset and between training and testing, the pre-processing was included within the cross-validation process. Then, a pre-processing model was saved to be used later to pre-process the test data and also to avoid the loss of information, as mentioned previously.

3.2 Classical classifiers

In this section are present the classical classifiers used in a first phase of the dissertation. The description of each classifier is based on scikit-learn's page.

AdaBoost

AdaBoost fits a classifier on the original dataset and then fits some copies of the classifier on the same dataset. Boosting is an ensemble technique that creates a strong classifier from a number of weak classifiers (models that are better than a random guessing).

The main principle of AdaBoost is to fit a sequence of models on different versions of the data. The predictions are then combined through a weighted sum to produce the final prediction. The production of this prediction is performed taking into account the application of different weights to each of the training samples, which initially they are initialized as $Wi = 1/N$. The first step is training a weak learner on the original data, then for each iteration, the sample weights are individually modified. During this process, those training examples that are incorrectly predicted at the previous step have their weights increased and for those whose the prediction was performed correctly, the weights are decreased. AdaBoost was the first boosting algorithm developed for binary classification.

The number of weak learners is defined by the number of estimators (*n_estimators*, default=50) and their contributions in the final result are controlled by the learning rate (*learning_rate*, default=1). Base estimators (*base_estimators*), maximum depth (*max_depth*) and a minimum samples per leaf required (*min_samples_leaf*) in decision trees are the main parameters responsible for the final results obtained with this classifier. The higher the number of minimum samples per leaf, the longer the time required for the classification process.

Snippet 3.1: Example of AdaBoost classifier

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import cross_val_score
n_estimators = 100
...
ada = AdaBoostClassifier(n_estimators=n_estimators)
```

DecisionTree

DecisionTree is a supervised learning method used for classification and regression. The main goal of decision trees is to create a model that performs a prediction of certain value of a target variable, learning simple decision rules according to the data features.

DecisionTree is a good classifier for visualization of the trees, which are simple to interpret and not requires a lot of data preparation. Unlike other techniques that are specialized in analyzing datasets of one type of variable, decision trees can handle numerical and categorical data and this enables to handle multi-output problems. The models can be validated using statistical tests, which may increase the reliability of the model. One of the main advantages of this classifier is its versatility, i.e., its logistic form can be used by programmers and engineers to diagnose medical failures in equipment.

Although the reliability of decision trees, its learners can create complex trees that may not generalize the data well, in other words, may originate overfitting. Hence, decision trees can be unstable due to the small variations in the data. The algorithms of this classical classifier are based on heuristic algorithms, but they may not guarantee to return the globally optimal decision tree. To solve this problem, features and samples can be randomly sampled with replacement and then the overfitting problem may disappear quickly.

X and Y are the inputs of this classifier. Array X of size [number of samples, number of features] holding the training samples, and array Y is of size [number of samples].

Snippet 3.2: Example of Decision Tree classifier

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
...

dt = DecisionTreeClassifier()
```

Nearest Neighbors

Nearest Neighbors classification is a type of non-generalizing learning. This type of classification stores instances of the training data and its classification is computed from a single majority vote of the Nearest Neighbors of each point.

Scikit learn has two types of Nearest Neighbors classifiers: **KNeighborsClassifier** and **RadiusNeighborsClassifier**. In the first one, the learning method is based on k (integer value specified by the user) nearest neighbors of each query point. The choice of k is important, because a larger k suppresses the effects of noise, but otherwise makes the classification boundaries less distinct. In the second, the learning method is based on the number of neighbors within a fixed radius r (floating-point value specified by the user) of each training point. For

the initial phase of this dissertation, has been decided to use the technique *KNeighborsClassifier* because is more commonly used than *RadiusNeighborsClassifier* technique.

The classification of Nearest Neighbors uses uniform weights. The value of query point is computed based on majority vote of the nearest neighbors. The default value for weights is ”uniform”, which assigns uniforms weights to each neighbor.

Snippet 3.3: Example of K Nearest Neighbors classifier

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score

num_neighbors = 10 # 5 is default
weight = 'uniform'
...
neigh = KNeighborsClassifier(n_neighbors=num_neighbors, weights=weight)
```

Logistic Regression

Although has ”Regression” in its name, Logistic Regression is a linear model for classification, not for regression. The probabilities of this model describes the possible outcomes of a single trial that are modeled by an logistic function. The implementation of Logistic Regression can fit binary or multinomial logistic regression with optional L2 or L1 regularization. L2 is a binary class. L1 regularization helps in feature extraction process in sparse feature spaces. This is why L1 regularization is used in some cases. Using L2 on the remaining variables can give better results in some scenarios than L1 by itself.

Logistic Regression is used in various fields, such as machine learning, medical field and social sciences. In Logistic Regression there are solvers. A solver is an optimization method to find the optimum of the objective function. There are four types of solvers: 'liblinear', 'newton-cg', 'lbfgs' and 'sag'.

The first one uses a coordinate descent (CD) algorithm. This algorithm implemented in liblinear cannot learn a true multinomial model, but its optimization problem is decomposed in a ”one-vs-rest” model. Hence, separate binary classifiers are trained for all classes. This type of solver is used in small datasets.

The remaining types of solvers only support L2 penalization and converge faster for high dimensional data. These solvers learns a true multinomial logistic regression model, i.e., its probability estimates may become better calibrated than ”one-vs-rest” model, which consists into fitting one classifier per class. The solvers ”lbfgs”, ”sag” and ”newton-cg” are used on large dataset, and for very large dataset only ”sag” is used.

Snippet 3.4: Example of Logistic Regression classifier

```
from sklearn import linear_model
from sklearn.model_selection import cross_val_score
...
lr = linear_model.LogisticRegression()
```

SVM

Support Vector Machines is a machine learning model, which is used for classification and regression. This model represents the samples as points in space, which are mapped into one side of a gap. This gap separates the categories attributed to each training sample by an SVM training algorithm. Then, the new samples are mapped into the same space and predicted to belong to a category, based on the side of the gap they are.

SVMs can be used in text and hypertext categorization, image classification and segmentation (SVMs achieve good search accuracy), recognition of hand-written characters, and in biological and science areas, mainly in protein's classification. In these areas, SVMs can also be used to identify features to perform predictions in some genetic datasets (such as performed in this dissertation).

SVMs are effective in high dimensional spaces, and in cases where the number of dimensions is greater then the number of samples in some datasets, but if the number of features is greater than the number of samples, the performance may be low. SVMs are also memory efficient, because they use a subset of training points in the decision function.

As the other classifiers, X and Y are the inputs of this classifier too. Array X of size [number of samples, number of features] holding the training samples, and array Y is of size [number of samples].

Snippet 3.5: Example of SVMs classifier

```
from sklearn import svm
from sklearn.model_selection import cross_val_score
...
clf = svm.SVC(kernel='linear')
```

Naive Bayes

Naive Bayes is a machine learning model, which its algorithms are based on Bayes' theorem with the "naive" assumption of independence between pair of features. In the past, Naive Bayes worked well in many real-world situations, mainly in document classification (sport, politic, etc) and spam filtering. Naive Bayes require small amount of training data to estimate

the needed parameters, and so this makes the classifier extremely faster compared to more sophisticated methods.

Naive Bayes can be considered as a simple technique for constructing classifiers. A family of algorithms is used, based on the assumption that the value of a particular feature is independent of the value of any other feature to train such classifiers. For the first phase of this dissertation it is used the Gaussian Naive Bayes algorithm, because for the type of classification intended (DNA classification), this algorithm gets more accuracy than the Multinomial and Bernoulli Naive Bayes algorithm.

X and Y are the inputs of this classifier. Array X of size [number of samples, number of features] holding the training samples, and array Y is of size [number of samples].

Snippet 3.6: Example of Gaussian Naive Bayes classifier

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_score
...
nb = GaussianNB()
```

Random Forest

Random Forest is a learning method for classification and clustering. Also, is an estimator that fits certain decision trees classifiers on several sub-samples of dataset. This classifier performs the averaging, which is a method to prevent the overfitting and hence improve the accuracy.

In this classifier, each tree is built from a sample from the training set. During the construction phase of the tree, the nodes are splitted and the split that is chosen is no longer the best split among the features.

Due to the randomness of this classifier, the bias of the forest may increase but considering the averaging, its variance also decrease. The bias is the error from wrong assumptions in the machine learning algorithms. A high bias can cause underfitting, which is the miss of relevant relations between features and target outputs.

The number of trees in the forest is determined by the number of estimators (*n_estimators*, default = 10) which each one of them have a maximum depth determined (if equals to None, the nodes are expanded until all leaves are pure, i.e., until it is not possible to expand further) and also have a maximum numbers of features (*max_features*), that determines the number of features to consider when looking for the best split (the tests performed with this classifier used a maximum features equal to None, which means that the max number of features is equal to the number of features of the dataset). As well as the other classifiers, X and Y are inputs of this classifier. X is size of [number of samples, num of features] holding the training samples, and Y is size of [number of samples].

Snippet 3.7: Example of Random Forest classifier

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

num_trees = 10
...
model = RandomForestClassifier(max_depth=None, n_estimators=num_trees,
    max_features=None, random_state=0)
```

Neural Network

Multi-layer Percepton (MLP) is a supervised machine learning classifier based on NNs. The algorithm of this classifier learns a function by training on a dataset and given numbers for input and output dimensions. Between the input and output layer, there can be one or more layers. This type of layers are called **hidden layers**. Each layer has also neurons inside them. Using this classifier, is possible to determine the number of hidden layers and neurons per layer.

The first layer, i.e, the input layer, consists of a set of neurons that represent the input features. Each neuron inside of a hidden layer performs the transformation of these values from the previous layer with a weighted linear summation *(w1*x1 + w2*x2 + w3*x3+ .... +wn*xn)*, followed by a non-linear activation function. The last layer, i.e, the output layer, receives the values from the last hidden layer and transforms them into output values.

This classifier has the capability to learn non-linear models and learn them in real-time. Its hidden layers have a non-convex loss function, and so, different random weight initializations may lead to different validation accuracy. This classifier is also sensitive to feature scaling and requires certain parameters such the number of hidden neurons, hidden layers and iterations (Figure 3.1 is based on [36]) .

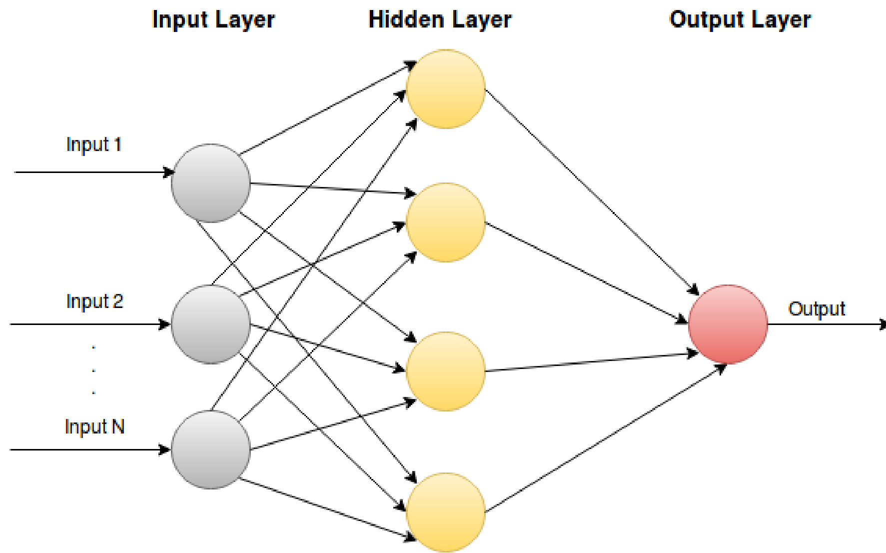


Figure 3.1: Structure of a simple neural network

In the input layer, the number of neurons corresponds to the number of inputs, which can be as many as needed. Then, each neuron of this layer performs a connection to each neuron inside the first hidden layer. The hidden layers can have neurons as many as needed too. After hidden layers, each neuron of the last one performs a connection to the output layer, which can have one or more outputs (depending on what we need to get). In this dissertation, neural networks have only one output, because it is pretended to get a score value between 0 and 1. MLP trains on two arrays: array X and array Y. Array X of size [n_samples, n_features], which holds the training samples and array Y of size [number of samples], which holds the class values for the training samples.

Snippet 3.8: Example of Multi-layer Percepton classifier

```
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_val_score

num_neurons_layer = 100
num_layers = 6

...
mlp = MLPClassifier(hidden_layer_sizes=(num_neurons_layer, num_layers),
    random_state=0)
```

3.3 Deep Learning

Deep Learning is a branch of machine learning based on a set of algorithms that modules high level abstractions in data. Deep learning provides the ability to derive predictive models without the need to know the underlying mechanisms and how they interact with each other, i.e., allows an abstraction of complexity. In biology's area, deep learning can be applied to predict if a genetic mutation may cause pathogenic effect or not, which is the main focus of this dissertation.

A recent advance in machine learning is the representation of the data with deep learning neural networks. Deep Learning NNs take the raw data at the lowest layer and transform them into feature representations by combining outputs with each other from the precedent layer in a data-driven manner, encapsulating all functions needed in the process [37]. NN is a supervised machine learning algorithm, which their training and classification tasks consist into use hidden layers and the respective nodes (will be explained in detail in chapter 4). Deep Learning has been increased significantly in the field of machine learning and has been used to improve speech recognition, image's performance and computational biology. In biology, the potential of deep learning is allowing to better exploit the availability of datasets (DNA sequencing, RNA measurements, etc.) by training complex networks composed by multiple layers. These networks discover high level features, improve performance, increase interpretability and provide understanding about the structure of the biological data [37]. In chapter 4 is performed and explained in detail one of these complex networks. To better understand Deep Learning, is essential to have some knowledge about Artificial Neural Network (ANN) and CNN, which will be explained below.

Artificial Neural Networks

Artificial Neural Networks consist of mathematical constructs, designed to approximate biological neurons, i.e., layers of interconnected neurons. The number of hidden layers corresponds to the depth and the maximum number of neurons in one of its layers corresponds to the width of neural network.

In configuration, presented in Figure 3.2, the network receives data in an input layer, and then are transformed through multiple hidden layers.

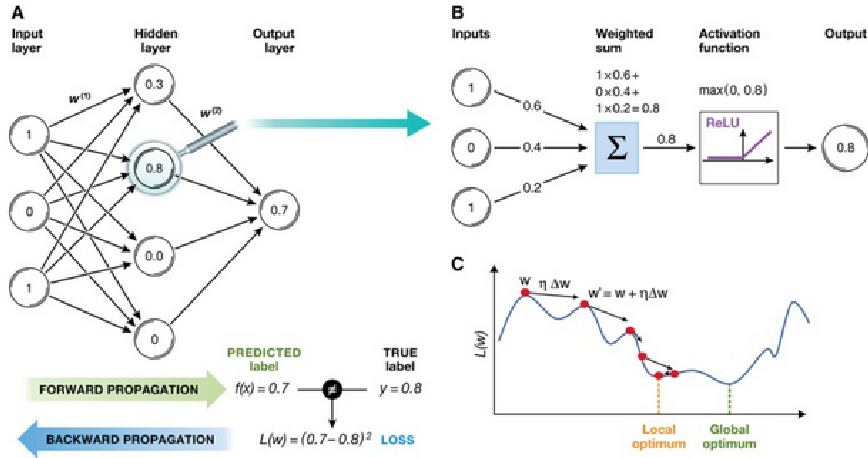


Figure 3.2: Workflow of artificial neural network [37]

The outputs are computed in the output layer after the network receives the data. A neuron computes the weighted sum of its inputs and uses an activation function to get the respective output *f(x)*, as shown in the panel B of figure 2.8. This activation function is the ReLU, which offers faster learning. This function limits negative signals to 0 and passes through positive signal [37] (explained further ahead). The weight $w^{1,2...,n}$ are parameters that capture the model's representation of the data. The loss function *L(w)* measures the fit of the model output to the true label of the sample (panel A). This function is minimized by the learning process. In each step of this function, the current weight vector (represented as red dot in graphic C) is moved along the graph in direction of the steepest descent Δw (direction arrows) by learning rate η (length of vector). During learning process, is possible to see in the panel A that the labels (*predicted* and *true*) are compared to compute a loss for the current set of model weights. The loss obtained is immediately backward propagated (represented with the blue arrow) to compute the gradients of the loss function. It is possible to jump over valleys at the beginning of the training and adjust parameters with smaller learning rates, only if the learning rates begin to decline over time [37].

Fully Connected Network

FCNN is a type of NN which has hidden layers that contain neurons and each neuron is connected to every neuron in the previous layer. Each connection has its own weight and may influence the learning process. FCNN makes no assumptions about the features in the data. FCNN is very expensive in terms of memory and computation, because of the large amount of connections and, such as stated previously, each connection has its own weight. Thus, this is why that generally do not exist a model composed only by a FCNN. FCNN is common used with convolutional layers in order to build a CNN model. In chapter 4 is explained in more detail the layers used by this type of NN.

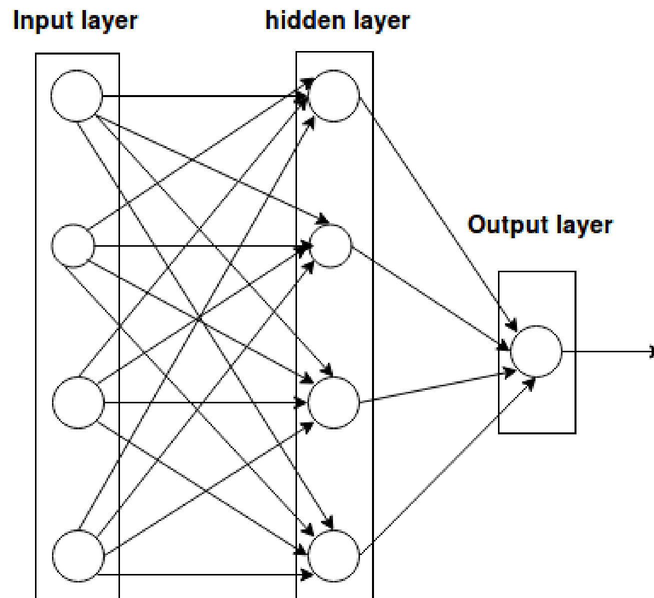


Figure 3.3: Fully Connected Neural Network [38]

Convolutional Neural Network

CNN was inspired by Hubel and Wiesel's work on the cat's visual cortex. They found that the cat had simple neurons that responds to simple motifs in the visual field. CNN is a type of artificial neural network in which the connectivity pattern between its neurons is based in animal visual cortex. A CNN consists of multiple convolutional and pooling layers. It allows learning more abstract features from small details, to object parts, and then the entire object.

CNNs began to be used in image pre-processing. In this area, it models input data in the form of multidimensional arrays: two-dimensional images with three color channels (RGB) or, in biology area, genomic sequences with one channel per nucleotide [37]. Due to high dimensionality of these data, the training is transformed into a FCNN as the number of parameters of such a model would exceed the number of training data to fit them. In order to solve this, CNNs make additional adjustments on the structure of the network. With this approach, the number of parameters are reduced.

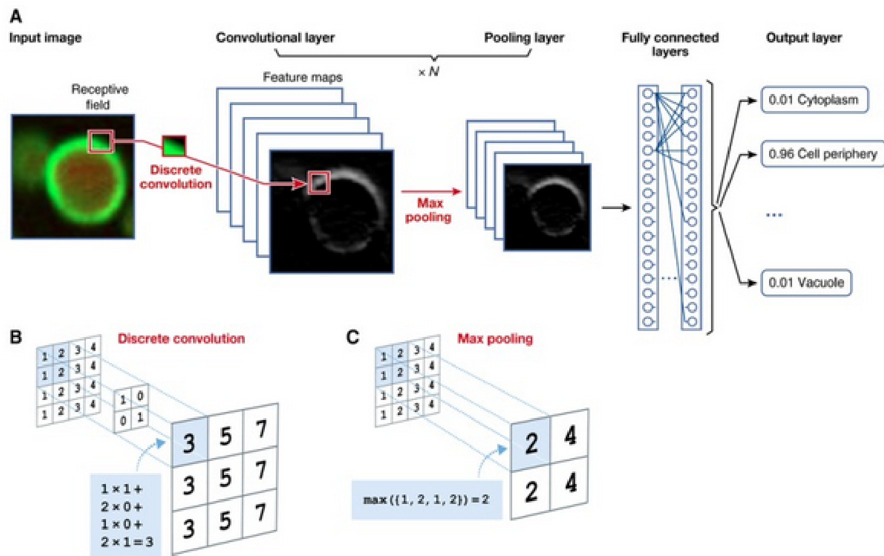


Figure 3.4: Workflow of convolutional neural network [37]

A convolutional layer consists of multiple maps of neurons, with their size equal to the dimension of the input image shown in panel A of Figure 3.4. The additional adjustments that can be performed on the structure of the CNN's network in order to reduce the number of parameters, can be based on two concepts, such as *local connectivity* and *parameter sharing*. First, each neuron within a feature map is only connected to a local set of neurons in the previous layer. Second, only the neurons within a feature map share the same parameters. These parameters are useful to compute a discrete convolution (panel B) of a neuron's receptive field, as represented in panel A of Figure 3.4, and then is computed the weighted sum of input neurons using an activation function, which is shown in panel B of Figure 3.4 [37].

In most applications using CNNs, some features such as position of objects and its format are irrelevant for the final prediction, i.e., the pooling layer aggregates adjacent neurons by computing the maximum over their activity (panel C). This results in a representation of feature activities and consequently a reduction on the number of parameters. Figure 3.4 shows the workflow of a CNN. All details of these networks and respective parameters are explained further ahead on chapter 4.

Deep Learning data

Every machine learning application requires pre-processing, feature extraction, training (only supervised machine learning) and testing data. Most applications of deep learning have been performed in supervised learning settings, because have labelled training samples available to fit several complex models. The number of training samples should be more than the number of model parameters [37].

When performing prediction molecular traits from genotype, the number of training instances are limited. One solution to circumvent this problem is considering sequence windows centred on the trait of interest [37]. This strategy is used in chapter 4 when our dataset is composed by the genetic sequence. An example in image analysis, labelled training examples may be difficult to obtain. In such instances, the training set can be augmented performing some operations (scaling, rotating). It is also possible to reuse a pre-trained network on a dataset for image recognition and adjust its parameters. In this approach, different datasets share important features (such as edges or curves).

All machine learning models must be trained, selected and tested on independent datasets, in order to avoid over-fitting and make the model generalizable to different data. The training set is essential to learn models with different parameters, assessed on the validation set. The selection of the model is based on its performance. The model with the best performance (for example prediction accuracy) is selected and evaluated on the test set, in order to compare with other methods. Tipically, 60% of dataset is for training, 30% is for validation and 10% is for model testing [37].

To predict molecular traits from DNA sequence, it is important to train models that use variation between regions within a genome. This is essential mainly because the prediction of the effects of rare mutations requires a large dataset. Even with these datasets, it is difficult to predict molecular traits from DNA sequence due to multiple layers of abstraction between the effect of DNA variants and the trait of interest [37]. In this context, deep neural network has two values. First, machine learning methods cannot operate on the sequence directly and require pre-defined features to be extracted from DNA sequence known a *priori* (features like conservation, motif occurrence, etc.). With deep neural networks is possible to avoid the extraction of features manually by learning them from data. Second, deep neural networks can get nonlinear dependencies in the sequence and interaction effects. Deep neural networks have been applied to specify DNA and RNA binding proteins [39] and to study the effect of DNA sequence mutations [40].

The first applications based on neural networks in this area replaced machine learning approach with a deep model with the same input features [37]. This model was trained using more than 1,000 features extracted from data. This method achieves a higher prediction accuracy of splicing activity compared with other approaches and identifies rare mutations with better accuracy.

First applications in computational biology

One of the early applications of deep neural networks had been related to pixel-level tasks using models on the network outputs. In the study [41] was trained a CNN on a 40 x 40 pixel matrix to classify the centre pixel to cell wall, cytoplasm and nucleus membrane using a CNN with three convolutional and pooling layers (explained in chapter 4), and a fully connected output layer. CNN obtained better results than other standard methods, such as Markov random fields. While CNN was trained, additional post-processing was required to get class probabilities from output images [37].

The addition of layers allows creating more abstract images, removing the noise of pixels. In the study [42] was performed an analysis of breast histology images in order to find mitosis in these images. This model used five convolutional and pooling layers that are followed by two fully connected layers, and with this won the mitosis detection challenge at International Conference of Pattern Recognition 2012 [37]. This approach was also to segment neuronal structures of microscopy images, performing a classification of each pixel as membrane or non-membrane. All models based in image classification, object detection, image retrieval and semantic segmentation use neural networks.

Deep learning frameworks: brief comparison

In order to perform pathogenicity prediction, it is essential to run the models over a deep learning framework. The most used deep learning frameworks are *Theano* (with Keras), *Caffe*, *Torch*, *TensorFlow* and *DeepLearning4j*. A good deep learning frameworks has good training and prediction speed (performance) and gets good classification accuracy. Based on work [43], is performed a comparison between these frameworks in terms of speed and classification accuracy.

The respective comparison is performed on a very used dataset in deep learning area. This dataset is called MNIST, which is a dataset of handwritten digits and is used for training image classification approaches. Below are shown examples of MNIST digit images.

Figure 3.5: MNIST digit images [43]

The digit images were inputted to the FCNN (explained in chapter 3). In the work [43] they used nearly 60 000 digit images as training set, and nearly 10 000 digit images as test set. The testing of NN was done using two activation functions: ReLU and Hyperbolic tangent (Tanh) (these functions are explained in chapter 4). An activation function is a function that maps input nodes to output nodes in a certain way, in order to make the training process faster.

To perform the respective assessment and comparison between these frameworks, has been used different methods. One of them is varying the number of hidden layers (”depth”) of network with fixed numbers of neurons per layer. In this method, the number of hidden layers vary from 1 to 4 and the number of neurons is 100. The other method is varying the ”width” (number of neurons) of network with fixed number of layers, which the number of neurons per layer has the following values: 64, 126, 512 and 1024. The Figure 3.6 and Figure 3.7 show the architecture of each method.

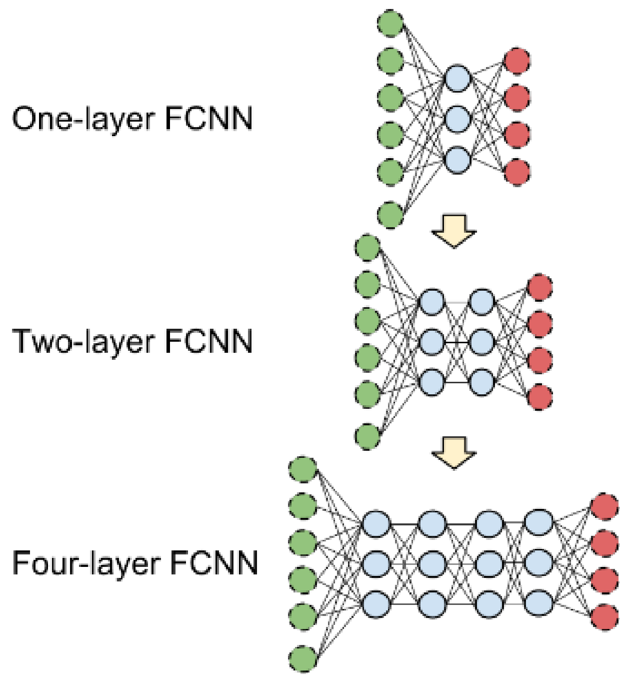


Figure 3.6: ”Depth” changes of NN architecture [43]

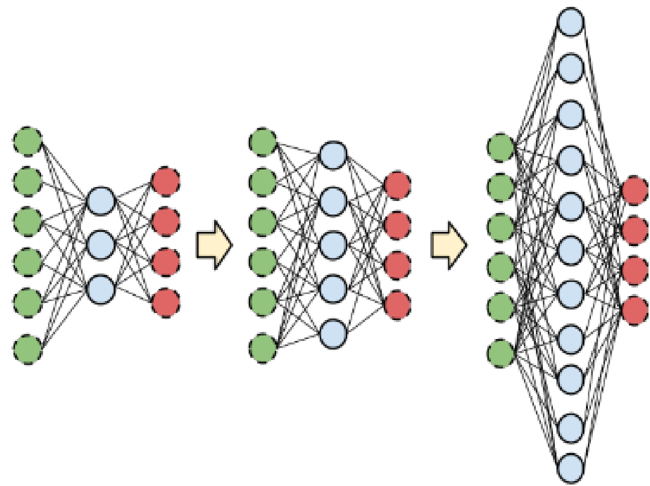


Figure 3.7: ”Width” changes of NN architecture [43]

Comparison results

Below are present the final results of comparison of all five frameworks by the FCNN training and prediction, and classification accuracy with ”Width” and ”Depth” architectures. Also, the number of epochs (the times that a model is trained) in all experiments is set to 10.

In first, is performed a comparison of training time between FCNN ”Depth” and ”Width” architectures:

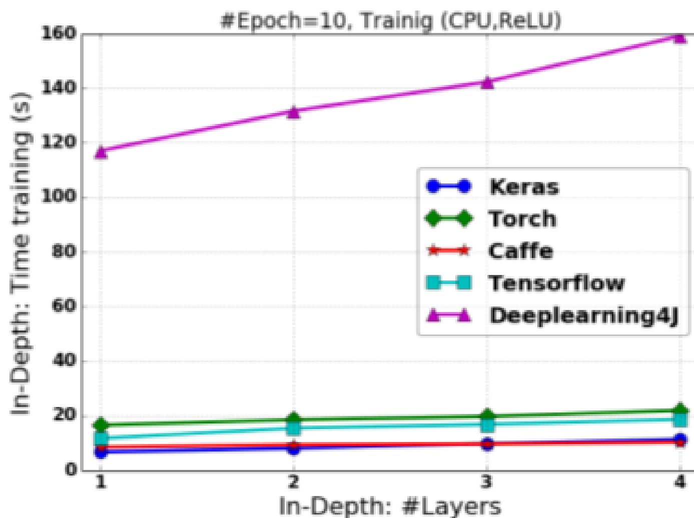


Figure 3.8: Training time for FCNN ”Depth” with ReLU function [43]

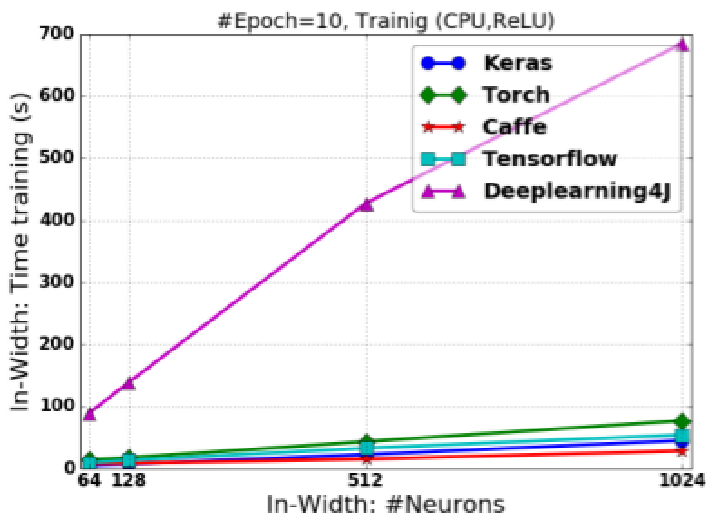


Figure 3.9: Training time for FCNN ”Width” with ReLU function [43]

The ”Depth” architecture training time for the frameworks start below from 20 seconds (1 hidden layer) and going up to nearly 30 seconds, except for DeepLearning4j which starts nearly with 110 seconds (1 hidden layer) and going up to 160 seconds (4 hidden layers). Relating to the ”Width” architecture, the training time for Theano (with Keras wrapper), Torch, TensorFlow and Caffe is always below of 100 seconds for 64, 128, 512 and 1024 neurons per layer. For DeepLearning4j framework, the training time starts with nearly 90 seconds and grows up to 700 seconds for 1024 neurons. These results are similar when using Tanh activation function. After all of these analysis about the training time results, we can conclude that the

”Width” architecture takes more time in the training process. It is possible to conclude that the testing time is similar between each architecture.

Relating to the testing time, the respective comparison is performed and present in Figure 3.10 and in Figure 3.11:

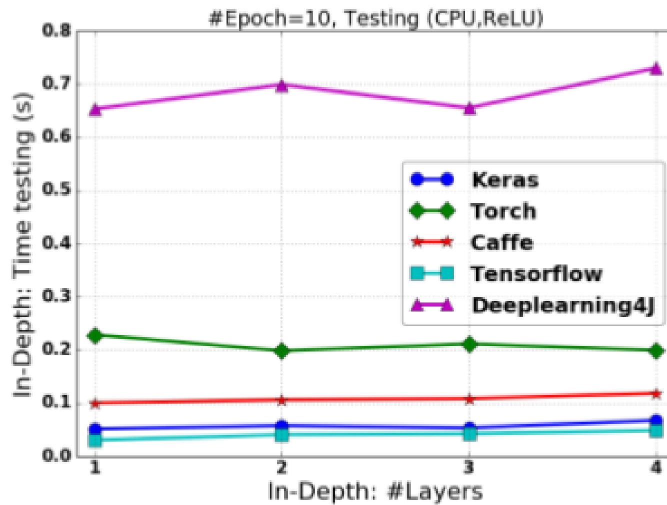


Figure 3.10: Testing time for FCNN ”Depth” with ReLU function [43]

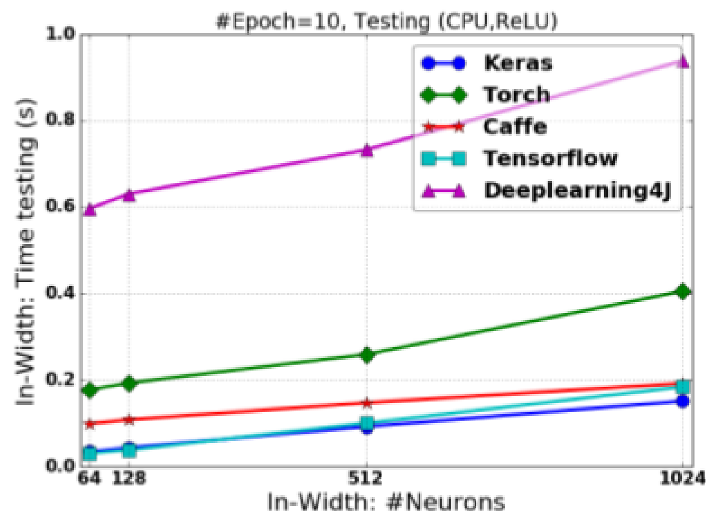


Figure 3.11: Testing time for FCNN ”Width” with ReLU function [43]

For ”Depth” architecture, the Torch framework takes nearly 0.2 seconds to test, Caffe often takes 0.1 seconds and the remaining frameworks have a testing time below 0.1 seconds except DeepLearning4j, which varies between 0.65 and 0.75 seconds. For ”Width” architecture,

DeepLearning4j starts with nearly 0.6 seconds (with 64 neurons) and grows up to 0.9 seconds with 1024 neurons. Torch starts with nearly 0.2 seconds for 64 neurons and grows up to 0.4 seconds for 1024 neurons. The remaining frameworks start with nearly 0.1 seconds for 64 neurons and grows up 0.2 seconds for 1024 neurons.

Finally, is performed a comparison of accuracy score between both architectures present below in Figure 3.12 and Figure 3.13:

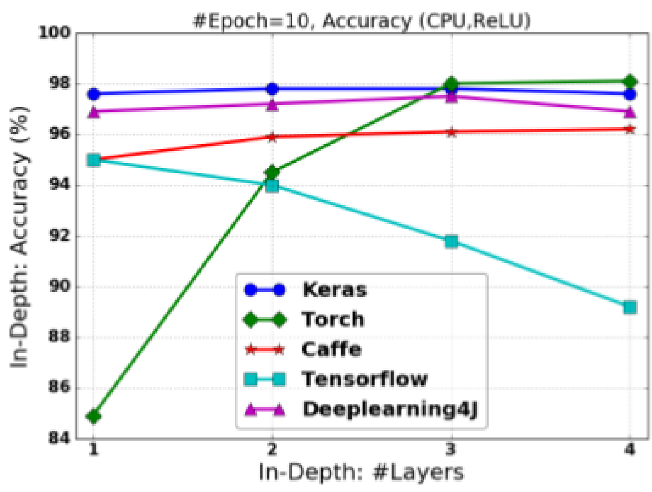


Figure 3.12: Accuracy classification for FCNN ”Depth” with ReLU function [43]

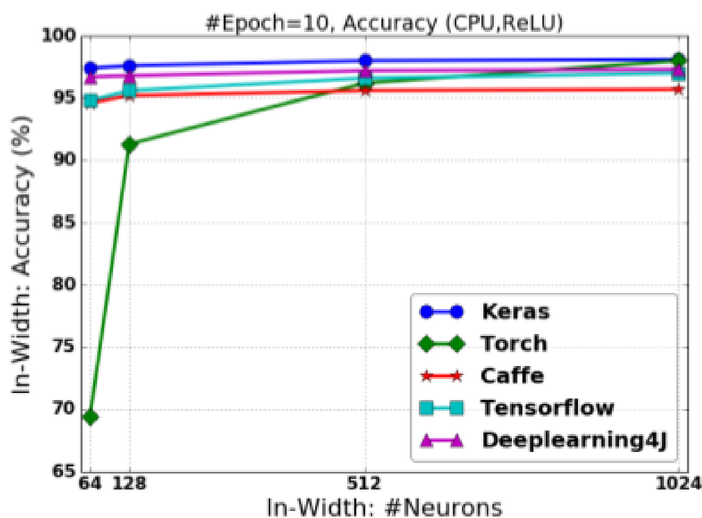


Figure 3.13: Classification accuracy for FCNN ”Width” with ReLU function [43]

For ”Depth” architecture, Torch framework starts with 85% (1 hidden layer) and goes to 98%. TensorFlow decreases from 95% (1 hidden layer) to 89%. Caffe grows up from 95% (1

hidden layer) to 96% (4 hidden layers) and the remaining frameworks are between 94% and 96%. Relating to ”Width” architecture, Torch starts to getting nearly 70% for 64 neurons, and then as the number of neurons increases, the accuracy increases too until more than 95% for 1024 neurons. The remaining frameworks obtained always more than 95% for 64, 128, 512 and 1024 neurons, such as the Figure 3.13 shows. It is possible to conclude that the accuracy score is similar between each architecture except for TensorFlow framework, which decreases when the number of hidden layers increase.

Frameworks	**Number of lines**	**Language**
Theano	*62*	Python
TensorFlow	*80*	*Python*
Caffe	*110*	*Protobuf*
DeepLearning4j	*130*	*Java*
Torch	*165*	*Lua*

Table 3.1: Framework complexity

Comparison conclusions

Increasing the number of hidden layers on a FCNN in order to improve the quality of classification increases the training time of the network, as well as increasing the number of neurons per layer. ReLU activation function increases network training speed and classification accuracy, unlike Tanh activation function.

In Torch and TensorFlow frameworks, the incorrect computation of its gradients during the network training led to the drop down of its classification accuracy, even the increasing of network depth. Other reason for this strange behavior might be the need of more sophisticated initialization of weighting coefficients [43]. The classification accuracy for these frameworks increased with increasing the number of neurons per layer. Increasing the number of neurons per layer, the accuracy of Caffe and DeepLearning4j frameworks dropped down. Accuracy of Caffe and DeepLearning frameworks dropped down, and the accuracy of Theano has been stable even with few neurons per layer.

It is important to note that DeepLearning4j was the lowest framework in the network training and prediction. The reason behind this is the fact that this framework is currently under development, and the training time, prediction time and accuracy may change in the future [43]. Considering the results obtained and the respective features, we could have a ranking list with the following order: Theano, TensorFlow, Caffe, Torch and DeepLearning4j, but has been chosen for this dissertation the TensorFlow framework due to have Google support.

3.4 Evaluation

To perform the evaluation, is required a collection of statistics derived from a confusion matrix. In the field of machine learning, a confusion matrix is a specific table that allows the visualization of the performance of an algorithm, in this case, the performance of a tool. A correctly classified test point is counted as a True Positive (TP) only if the test point corresponds to the positive class (pathogenic or damaging) and as a True Negative (TN) only if the test point corresponds to the negative class (neutral or benign). Thus, False Positive (FP) is a negative test point that is classified to be positive, and a False Negative (FN) is a positive test point classified as a negative one [3]. These datasets are unbalanced, thus the performance of single tools was assessed by computing Receiver Operating Characteristics (ROC) curves and Precision-Recall curves (ROC-PR-curves) [3]. The first one is the fraction of the TP over all positives TP+FN against the fraction of the FP over all negatives TN+FP (1-specificity or FP rate), and the second (ROC-PR-curves) is the fraction of the TP over all positives TP+FN (sensitivity or recall) against the fraction of the TP over all TP+FP (precision) [3]. To obtain the performance, is computed the area under ROC-curves and ROC-PR-curves (AUC - Area Under Curves and Area under the receiver operating characteristic curve Precision-Recall (AUC-PR) - Area Under Curves Precision Recall), in which the last one takes values between 0 and 1 [3]. Perfect classifiers have an AUC and AUC-PR equal to 1, and random classifiers have an AUC and AUC-PR equal to 0.5 [3].

To help realize better what was said previously, are shown the fractions previously mentioned [3]:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall/Sensivity = \frac{TP}{TP + FN}$$

$$Specificity = \frac{TN}{FP + TN}$$

$$NegativePredictValue = \frac{TN}{TN + FN}$$

These formulas are used to obtain the results for each metric used in deep learning models on the datasets. For this dissertation, are used 3 metrics: Accuracy, AUC (ROC Area under curve) and MCC (Matthews correlation coefficient). These are the metrics used, because they work well in binary classification, which is the type of classification pretended. The metrics are used to measure classification performance and accuracy of classification model. Below is

shown a description of each one:

- **Accuracy**: the accuracy is computed by *accuracy_score* function, either the fraction or the count of correct predictions. It is the number of correct predictions divided by the total number of predictions made (multiplied by 100 to turn it into percentage). When performing multilabel classification, *accuracy_score* returns the subset accuracy. If the set of predicted labels for a sample exactly matches with the true set of labels, then the subset accuracy is 1, otherwise is 0.

- **AUC**: ROC Area under curve is computed by *roc_auc_score* function, which as the name states, computes the area under the curve from prediction scores. This metric is applied also in binary and multilabel problems. This metric uses a confusion matrix which contains a count of TP, FN, FP and TN. Then, is performed a briefing in order to know how many data points was correctly and not correctly classified. Once these values are acquired, is computed the True Positive Rate (TPR) and the False Positive Rate (FPR), because these values are then combined into one single metric with many different thresholds for the logistic regression. The resulting curve is called ROC curve and the metric considered is the AUC of this curve, as the Figure 3.14 shows:

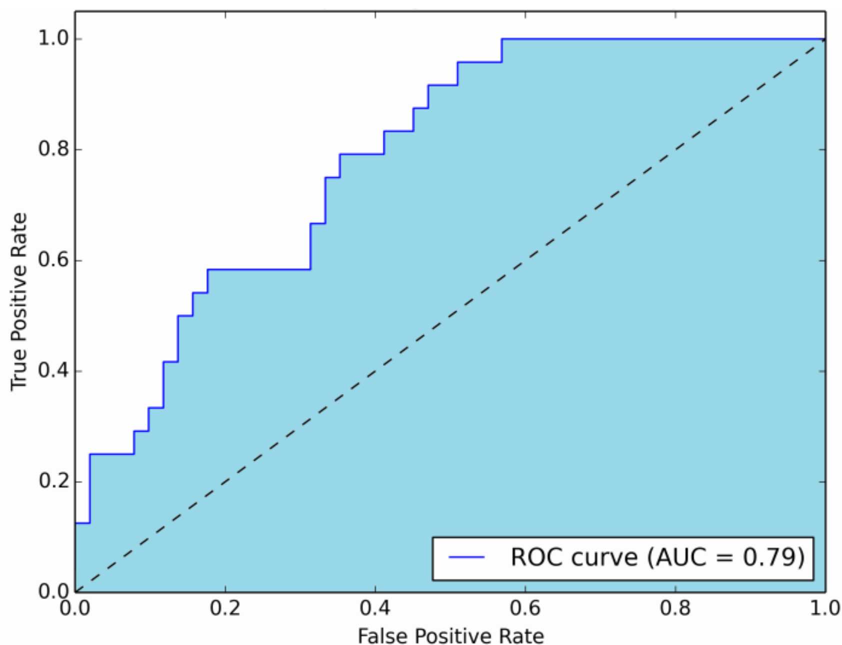


Figure 3.14: Graphic representation of ROC_AUC metric [44]

The blue area in Figure 3.14 corresponds to the Area Under the curve of Receiver Operating characteristics (ROC curve and the dashed line in the diagonal presents the

ROC curve of a random predictor which has a ROC curve of 0.5).

- **Matthews correlation coefficient (MCC)**: Matthews correlation coefficient is computed by *matthews_corrcoef* function for binary classes. This function is used in niary classifications and also takes into account the true and false positives and negatives, and is considered as a balanced measure which can be used even if the classes have different sizes. The correlation coefficient value is between -1 and 1, which 1 represents a perfect prediction, 0 represents an average prediction, and -1 represents an inverse prediction.

3.4.1 Train and test

This project is focused on classification, in order to predict if an mutation in a genetic sequence can cause pathogenicity or not. Scikit-learn's (described in chapter 4) classical classifiers (stated previously) performs this prediction but before is necessary to train and test the model. Initially, the data is splitted into two sets: train and test. Tipically, 70% of the data is for training and the other 30% is for testings, but this can be modified in function *train_test_split* by the parameter **test_size** before applying the respective classifier.

Snippet 3.9: Definition of train and test set

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
    random_state=0)
```

In the snippet above, the variable X represents the training samples, y holds the class labels for the training samples, the test size represents the percentage of the data to test (evaluate, testing set) and the variable random state represents the randomness of the final result, i.e, if the test is equal to some number, then the split will be always the same, thus the result will be always the same too.

After training the model on the dataset, is performed the respective classification (test). The classification process uses metrics, which are functions that assesses prediction error for some purposes. They are used to measure classification performance and accuracy of classification model (as already seen previously).

One mistake that is usually performed is learning the parameters of a prediction function and testing it on the same data. A model to which repeats the labels of the samples gets a perfect accuracy, but fails to predict anything useful on yet-unseen data. This situation is called overfitting. To prevent it, it is a common practice to hold out part of the available as a test set *X_test, Y_test* (used in the snippet above). Other way to avoid this situation is to use **cross-validation**, which is explained below.

3.4.2 Cross-validation

Cross-validation is a model validation technique, which is used to know how the results of statistical analysis on the dataset will generalize to an independent dataset. The main goal is define a dataset to test the model in the training process, in order to avoid the situation of overfitting. Other reason to use cross-validation instead of using is that there is not enough data available to partition it into training and test sets without losing testing capability.

Initially in this dissertation, cross-validation was used to improve the final accuracy, using strategies from one type of cross-validation. The type used is non-exhaustive, because the methods of this type do not compute all ways of splitting the original sample, hence, the final result is better than exhaustive type.

In this cross-validation type there is a method that splits the data randomly into K equal sized subsamples. The method is called **k-fold cross-validation**. Of these K subsamples, a single subsample is chosen as the validation data to test the model, and the other K-*1* subsamples are the training data. The process is then repeated K times (K is the number of the folds chosen), which each one of the K subsamples are used once as validation data (test).

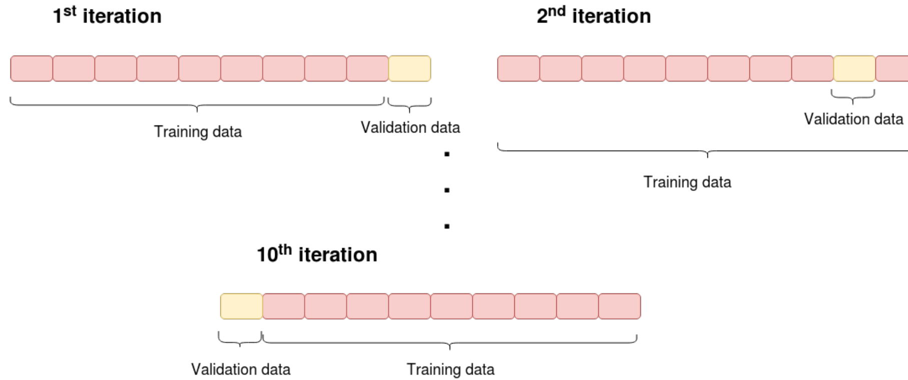


Figure 3.15: Application of 10-fold cross-validation

In the end of this process, the K results from the folds are averaged to produce a single estimation of the accuracy. The main advantage of this method, and that explains why this method is the most used by researchers, is that all observations are used to training and validation (as the figure above shows), which each of them is used once as validation data. Tipically, the most used value for K is 10, i.e, 10-fold cross validation, which means that 9 subsamples are the training data and the remaining are the validation data. As said before, the process is repeated, in this case, 10 times which each subsample is used once as validation (testing) data.

This method allows to apply model selection, in order to improve the accuracy. In this dissertation, it is applied two of the most used models selection:

- **Stratified K-fold:** is a variation of *K-fold* which each set has the same percentage of samples of each target class. This type of model selection is helpful when we have large imbalance in the distribution of the target class, i.e, there might be more negative than positive samples. The relative class frequencies are preserved in each train and validation fold. As the name states, this model selection returns stratified folds (the snippet below is based from scikit-learn's page.

Snippet 3.10: Application of Stratified K Fold model selection

```
from sklearn.model_selection import cross_val_score, Stratified K-fold

x = np.array(1.3, 4.5, 2.3, 6.7, 5.6, 7.8, 9.2, 3.4, 2.1, 5.1)
y = [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
skf = StratifiedKFold(n_splits=3) #n_splits specifies the number of folds
for train, test in skf.split(x, y):
   print("%s %s" % (train, test))

#print the indexes of training and testing data
[2 3 6 7 8 9] [0 1 4 5]
[0 1 3 4 5 8 9] [2 6 7]
[0 1 2 4 5 6 7] [3 8 9]
```

- **Group K-fold:** is a variation of *K-fold* which states that the same group is not present in both training and testing sets. This can be applied for instance when we have medical data from multiple patients, in which these data have multiple samples collected from each one, and such data is dependent from individual group. In this case, the identifier (ID) of the patient for each sample will be its group identifier (in this dissertation, the identifier is the first column of the dataset, i.e., the gene ID from an individual). From here, the model will train on a particular set of groups, in order to generalize well to the unseen groups, but it is essential to ensure that all the samples in the validation (testing) fold come from groups are not represented in the training fold. Note that the resulting folds do not have the same size, due to the imbalance of the data, as it occurs in Stratified K-fold model selection (the snippet below based on scikit-learn's page).

Snippet 3.11: Application of Group K Fold model selection

```
from sklearn.model_selection import cross_val_score, Group K-fold
x = np.array(1.3, 4.5, 2.3, 6.7, 5.6, 7.8, 9.2, 3.4, 2.1, 5.1)
y = ["gene1", "gene2", "gene2", "gene2", "gene3", "gene3", "gene3",
    "gene4", "gene4", "gene4"]
groups = [1,1,2,2,3,3,3,3,3,3]
```

```
groupKf = GroupKFold(n_splits=3) #n_splits specifies the number of folds
for train, test in groupKf.split(x, y, groups=groups):
    print("%s %s" % (train, test))
#print the indexes of training and testing data
[0 1 2 3] [4 5 6 7 8 9]
[0 1 4 5 6 7 8 9] [2 3]
[2 3 4 5 6 7 8 9] [0 1]
```

CHAPTER 4

Pathogenicity prediction

The chapter 3 presents the machine learning concept, its methods, models used to perform predictions, its sub-field Deep Learning, which contains deep models to perform pre-processing, feature extraction, training and classification on the datasets. This chapter's objective is implement models that perform a training and classification on the created models using the DNA sequence, in order to perform a prediction and finds out if an mutation occurred in the sequence is pathogenic or not. Each step performed in the implementation and the reasoning behind each decision will be explained in detail, taking into account all machine learning concepts approached in chapter 3. For this purpose, some diagrams, tables and code snippets will be used to show some examples of using some tools/classifiers.

The chapter starts by defining the objectives to be accomplished in section 4.1. Then, a description of each technology and library adopted is performed in section 4.2, with Scikit-learn and its classifiers, Keras NN, deep learning framework and respective reason of choose. After that, we start to explain what was done step by step in each phase of this dissertation.

In section 4.3 are defined the datasets used in this dissertation. Then, in section 4.4 are explained the steps performed when using the scikit classical classifiers and the reasons of each decision. In section 4.5 are explained the different encoding strategies used on the data from the used datasets to perform the training and classification on the models. Also, in this chapter, in section 4.6 are present the models built to perform the training and classification, taking into account the scoring metrics used (*AUC*, *Accuracy* and *Matthews Correlation Coefficient*).

4.1 Objectives

As was stated in this dissertation, the main goal is to create models based on *deep learning* that allow to define if a given mutation in genetic code can or cannot cause pathogenic effects. Another goal is to know what are the best models for prediction, based on the results of scoring metrics (Accuracy, ROC_AUC and Matthews correlation coefficient) and the performance.

To perform this work, it is required to know which classifiers are available to perform the classification and testing on the available datasets. In the chapter 2 have been mentioned some datasets, which five of them will be used to perform data pre-processing, classification and testing. The datasets are Exovar, Humvar, PredictSNP, Swissvar and Varibench. A brief definition of their format is explained in section 4.3.

Then, in section 4.4 is explained the steps to perform predictions on the five used datasets, which have a lot number of samples and features obtained by mathematical functions using the scikit classical classifiers (explained in section 3.2). After that, is explained the encoding strategies used to encode the data that is in string format.

Lastly, in section 4.6 is explained all steps performed in the building of the models based on deep networks. All the data, using each encoding strategy, will train these models to perform the respective pathogenicity prediction, by classifying the data.

4.2 Technologies and libraries

4.2.1 Scikit-learn

Scikit-learn is a python free machine learning Application Programming Interface (API) that features a lot of classification, regression and clustering algorithms. This library has a lot of classifiers such as SVM, Random Forest, AdaBoost, DecisionTree, etc, which they are designed to incorporate with *Numpy* and *SciPy* libraries. Scikit-learn emerged in the Google Summer of Code project, which was written using Python, Cython and C++.

Scikit-learn started to be used with small datasets, which were created to illustrate the behaviour of the various algorithms implemented by the Scikit-learn. These datasets were also created by Scikit-learn and the main goal is to get an accuracy, applying different techniques of classification, regression and clustering on the data. Some datasets available by Scikit-learn and most used are: **iris** (classification), **diabetes** (regression), and **digits** (classification).

Scikit-learn can be accessed and installed in the following link: `http://scikit-learn.org/stable/install.html`

4.2.2 TensorFlow

TensorFlow is an open-source software library, developed by Google for Machine Learning, for systems capable of building and training neural networks, using several different datasets. Initially, the main goal of this open-source software library was to detect and decipher patterns and correlations, similarly to the learning and reasoning of humans.

TensorFlow was released in November, 2015. As opposed to the reference implementation, TensorFlow can run on multiple Central Processing Unit (CPU) and Graphics Processing Unit (GPU), and also with optional Compute Unified Device Architecture (CUDA) extensions. TensorFlow is available on 64-bit Linux, Mac OS and mobile computing platforms such as Android and iOS. The name of this open-source software library derived from the operations that neural networks perform on ”tensors”, that are multidimensional data arrays.

TensorFlow uses nodes and graph edges, due to the fact that is designed for numerical computation. The nodes represent mathematical operations and the graph edges represent the tensors communicated between them. Also, this framework improves efficiency and modularisation in distributed computation. As mentioned before, TensorFlow allows to build graph-based models, such as those used in machine learning and artificial intelligence, and those models may run on a distributed computing systems. Nowadays, TensorFlow is used by a lot of know companies, such as DropBox, ebay, Google, Intel, Snapchat, Twitter, etc.

Despite of there are other frameworks to work with deep learning such as **Theano** and **Torch**, TensorFlow was chosen to work in this dissertation because companies as Google offers support to TensorFlow. Google is offering less support for Theano and Torch libraries.

Thus, Google will migrate Deep Mind away from Torch and Theano. This is the reason why deep learning users have been using more the TensorFlow's framework. In this dissertation, TensorFlow runs in backend of **Keras** (explained further on) when the pre-processing, train, and classification (test) are performed.

4.2.3 Keras

Keras is an high-level and open source neural network API, written in Python that can run on top of TensorFlow and Theano. Enables fast execution with deep neural networks, focusing on being modular and extensible.

Keras allows easy and fast prototyping, i.e, it is a user friendliness API. Keras offers consistent and simple API, minimizes the number of user actions required for some common tasks, because it is an high-level API that runs on top of TensorFlow, i.e, to perform an action in TensorFlow, is required to perform some other tasks before the execution of the action, and in Keras these tasks can be performed with a simple line of code. This is main reason that in this dissertation it was used Keras instead TensorFlow directly. Keras may also provide error feedback to the user.

In Keras, the modules can be combined with few little restrictions. **Neural layers**, **cost functions**, **optimizers**, **initializations schemes** and **activation functions** are standalone modules that can be combined to create new models. This is why Keras has the principle of modularity. These new modules are simple to add, hence, builds up easily a model that makes Keras suitable for advanced research. Models are described in Python, which is compact, extensible, easier to debug and quick construction.

As we can see, the **model** is a way to organize and create layers. The simplest type (and used mostly in this dissertation) is the **Sequential** model, that is an organized set of linear layers (Keras also has models which result in a combination of other models with other types of layers, resulting in a complex deep neural network). With this model, is possible to make a **Fully Connected Network** with Fully Connected layers, and a **Convolutional Neural Network** with Convolutional and Fully Connected Layers. The following headings are a bit based on Keras web page.

Fully Connected layers

Fully Connected layers enable the construction of a Fully Connected Network. In this network, a neuron of each layer is connected to every neuron in the previous and next layer, and each connection has a given weight. This network makes no assumptions about the features in the data, and it is very expensive in terms of weights (too much used memory) and connections (computation tasks).

Snippet 4.1: Example of an simple Fully Connected Network

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(32, input_shape=(64, )))
```

```
model.add(Activation('relu')) # equivalent to model.add(Dense(32, input_shape(64,
    ), activation='relu'))
model.add(Dense(32))
model.add(Activation('softmax'))
```

In the snippet above, it is created a Sequential model which will take as input arrays of shape (*, 64) and output arrays of shape (*, 32). It is common to use a kernel initializer parameter, which is the initializer for the kernel weights matrix. The initialization defines the way to set the initial random weights of Keras layers. After this, is necessary to add an activation function, because it turns the train more faster and is not necessary add more parameters to the model. There are several kinds of activation function, such as *softmax*, *relu*, *sigmoid*, *tanh*, etc. Considering the main purpose of this dissertation, it will be used the *relu* activation function, because it is faster to train the network even without pre-training than the other activation functions, and does not face with vanishing gradient problem (it is a difficulty found in training certain Neural Networks with gradient based methods) as with *sigmoid* and *tanh* functions. Then, after the first layer is no need to specify the size of input anymore, because the neural network knows it already. The most used types of activation functions are *softmax*, *relu*, *sigmoid* and *tanh*:

- **softmax:** Is a function that squashes the outputs of each unit between 0 and 1 and it is used in multi-class classification problems (for example on MNIST dataset). Each output is divided such that the total sum of the outputs is equal to 1.

- **ReLU:** The ReLU activation function is defined as $f(x) = max(0, x)$, where x is the input to a neuron. It is the most popular activation function for deep neural networks, due to its capacity to train faster (six times than tanh function) than the other functions. This function has efficient gradient propagation, i.e, has no vanishing gradient problems, has efficient computation because only has comparison, addition and multiplication operations and it is faster to train even in large and complex datasets. ReLU has also soft plus units superseded from soft plus function because it turns the computation faster and linear.

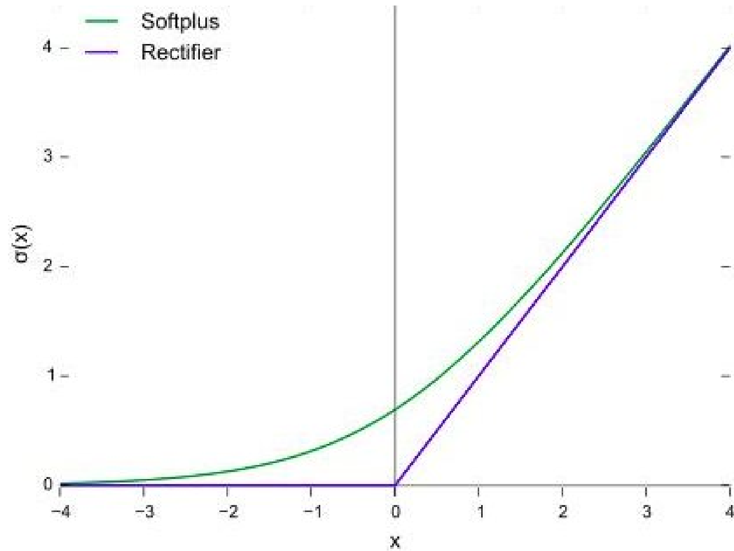


Figure 4.1: ReLU function and softplus [45]

- **sigmoid:** Sigmoid function is defined as $S(x) = 1 / (1 + e^{-x})$ and returns values in range [0, 1]. This activation function is mostly used in binary classification (as we pretend to do in this dissertation). Sigmoid is an exponential function and this is one of the main reasons that is one of the most used, because generally are similar to handle mathematically, and computationally is simple, which offers better performance for learning algorithms. In this dissertation, this function is used in the last layer of fully connected network, which receives some inputs and has only one neuron (output), in order to know the final classification that is a value in range [0, 1].

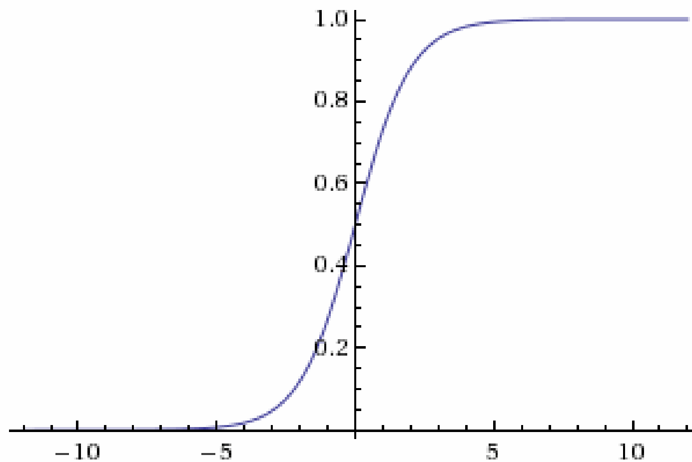


Figure 4.2: Sigmoid function [36]

- **tanh:** Tanh function is a sigmoid re-scaled function, because its output range is [-1,1], instead [0, 1]. Is the hyperbolic tangent function, which is similar to the Tan circular function used throughout trigonometry. This activation function is expensive to compute on most architectures and most of time is quickly converge than sigmoid and performs better accuracy.

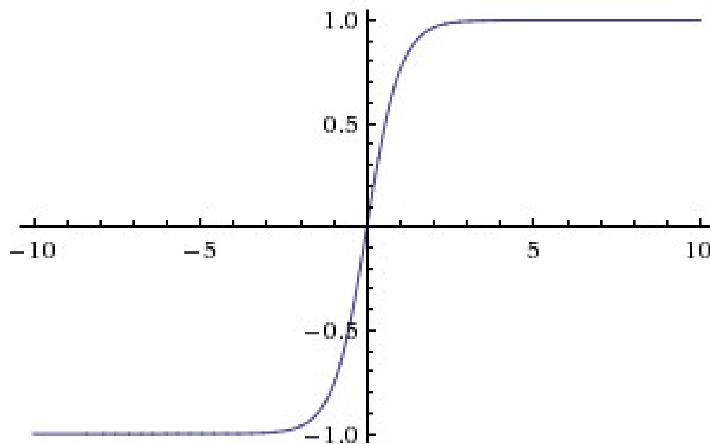


Figure 4.3: Tanh function [36]

Keras has also wrappers for the Scikit-learn API, which allow to use sequential Keras models as part of our Scikit-learn workflow. There are two wrappers for the Scikit-learn API: *KerasClassifier* and *KerasRegressor*, but we only pretend to perform classification, so it will used *KerasClassifier*. KerasClassifier commonly uses a class instance to construct, compile and run the Keras model (*build_fn*) and also takes the model and fitting parameters (*sk_params*). This parameter accepts parameters for calling *fit*, *predict*, *predict_proba* and *score* methods such as *epochs*, which separates training into distinct phases, and batch size, which is a set of N samples that are processed independently in parallel. The larger the batch, the better the approximation, but may take longer to process. To beyond these parameters, model and fitting parameters accepts also a verbose, which may show or not the progress bar logging for epochs (0 for no logging, 1 for progress bar logging and 2 for one log line per epoch).

Convolutional Neural Network and layers

In this section, is discussed pratical concepts of a CNN based on Keras API. As was already stated in chapter 3, CNN is based on animal visual cortex.

These type of NN uses **Convolutional layers**, which are the essential pillar for CNNs. These layers are the core building block of a CNN and creates a convolution kernel that is involved with the layer input over a spatial dimension in order to produce a tensor of outputs. The first convolutional layer on the model requires the specification of an *input_shape* argument

which generally is a tuple of integers. There are several types Convolutional layers, but the 3 most common are (one of them is used in this dissertation) Conv1D, Conv2D and Conv3D.

Conv1D layers are mostly used in problems like text classification and, for the purpose of this dissertation, DNA sequence classification. These layers receives a certain number of samples from a dataset, which has a defined number of features and each one are multidimensional.

Snippet 4.2: Convolutional 1D layer

```
from keras.layers import Conv1D
from keras.models import Sequential

kernel_size = 3

model = Sequential()
model.add(Conv1D(filters=32, kernel_size=kernel_size, input_shape=(16, 256),
    padding='same', activation='relu'))
```

The snippet above shows the creation of the model and then it added an Convolutional layer. The argument *filters* is the dimensionality of the output space, i.e, the number output of filters in the convolution. The *kernel_size* argument in Conv1D layer is an integer which specifies the length of the 1D Convolution window. The *input_shape* argument defines the shape of the input, i.e., the number of vectors and its dimension. In the case of this snippet we have can have any number of samples, and specified in the argument, sequences of 16 vectors (features) of 256-dimensional vectors. The padding is also important because it is easier to design networks, preserving the height and width (no need to worry about tensor dimensions), and then the network can easily become a deep network, which improves the performance by keeping the information in the borders. In the snippet, *padding = "same"* means that there is no changes tensor dimensions and in height and width. Finally, we have the activation function applied to this layer. The activation function applied is ReLU, which was already explained.

Such as the Conv1D layers, Conv2D layers are also essential to build a CNN, but these layers are used mostly in image pre-processing. In the early days, deep learning started to work with images. Images have dimensions (height and width), pixels and channels (such as Red Green and Blue (RGB), grayscale), so is needed to add an additional value for the shape of input.

Snippet 4.3: Convolutional 2D layer

```
from keras.layers import Conv2D
from keras.models import Sequential

model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3,3) input_shape=(128, 128, 3),
```

```
    padding='same', activation='relu'))
```

In the snippet above, the convolutional layer accepts image with the dimension 128x128 in RGB format (the value 3 specifies each channel of the images). The kernel size is now a tuple, because in addition to having the number of channels, we have width and height.

Conv3D layers are also used to create an CNN. These layers are little used because it makes the network slower and there are a few datasets with 3 dimensions. In this layer, the shape of input is for 128x128x128 volumes with 3 channels.

Snippet 4.4: Convolutional 3D layer

```
from keras.layers import Conv3D
from keras.models import Sequential

model = Sequential()
model.add(Conv3D(filters=32, kernel_size=(3, 3, 3) input_shape=(128, 128, 128, 3),
    padding='same', activation='relu'))
```

In a CNN is usual to add periodically a **Pooling layer** between successive Convolutional layers in a Convolutional Network, in order to reduce the spatial size of the representation, which reduces the amount of parameters and computation in a network. Basically, it is an essential layer to control the overfitting problem. This layer works in each depth slice of the input and resizes it spatially, using the **MAX** function. That's why in this dissertation is used the **MaxPooling1D** layer.

Snippet 4.5: MaxPooling 1D layer

```
from keras.layers import MaxPooling1D
from keras.models import Sequential

model = Sequential()
model.add(MaxPooling1D(pool_size=2))
```

The snippet shows the application of the size of maximum pooling windows in a Convolutional 1D Neural Network (*pool_size*=2). The figure below explains how a MaxPooling layer 2D works in each slice depth. The value of the size of pooling is the factor by which to downscale. In this case, a size of pool equal to (2,2) will halve the input in both dimensions.

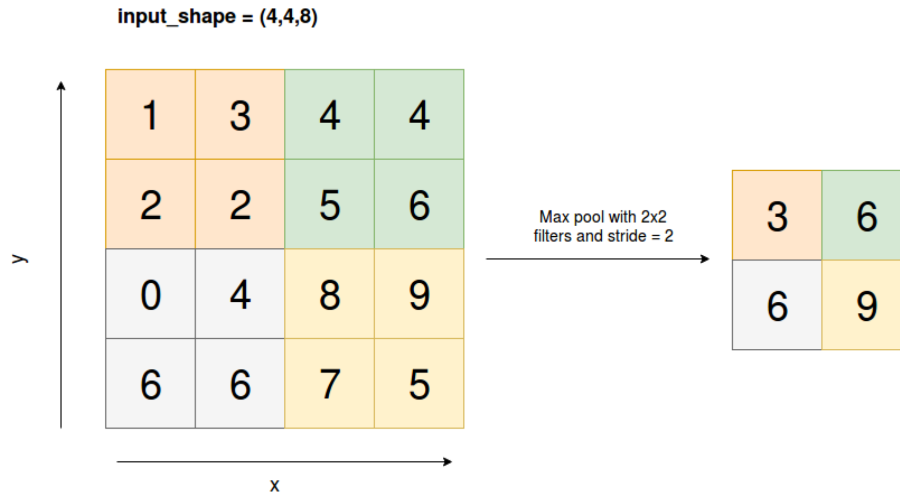


Figure 4.4: MaxPooling layer operation [36]

The most usual form is a pooling layer with 2x2 filters, which are applied with a stride of 2 downsamples, such as represented in Figure 4.4.

Snippet 4.6: MaxPooling 2D layer

```
from keras.layers import MaxPooling2D
from keras.models import Sequential

model = Sequential()
model.add(MaxPooling2D(pool_size=(2,2))
```

Snippet 4.7: MaxPooling 3D layer

```
from keras.layers import MaxPooling3D
from keras.models import Sequential

model = Sequential()
model.add(MaxPooling3D(pool_size=(2,2,2))
```

Such as MaxPooling1D and MaxPooling2D layers, MaxPooling3D layers have also have the size of pooling. As shown in the snippet above, a size of pool equal to (2,2,2) will halve the size of input in each dimension.

After inserting a MaxPooling or a Convolutional layer, is possible to add a **Dropout**. Dropout is used to help in the prevention of overfitting by setting a fraction rate of input units to 0 at each update in training process.

Snippet 4.8: Application of a Dropout with a rate of 0.2

```
from keras.layers Dropout, Conv1D, MaxPooling1D
from keras.models import Sequential

kernel_size = 3

model = Sequential()
model.add(Conv1D(filters=32, kernel_size=kernel_size, input_shape=(16, 256),
    padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.2))
```

We can insert Convolutional, MaxPooling and Dropout layers how many we want, but in order to perform a complete Convolutional Neural Network is needed to connect all these layers with a Fully Connected Network. This important connection is performed by the **Flatten** layer.

Snippet 4.9: Application of Flatten layer

```
from keras.layers Dropout, Conv1D, MaxPooling1D, Flatten
from keras.models import Sequential

kernel_size = 3

model = Sequential()
model.add(Conv1D(filters=32, kernel_size=kernel_size, input_shape=(16, 256),
    padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.2))
model.add(Flatten())
#fully connected layer
model.add(Dense(32, input_shape=(64, )))
model.add(Activation('relu'))
...
```

After inserting the Flatten layer, we can add the Fully Connected Network in order to create the Convolutional Neural Network. But before training the model, it is essential to configure the learning process. This is done by the method **.compile()**.

Snippet 4.10: A complete Convolutional Neural Network

```
from keras.layers Dropout, Conv1D, MaxPooling1D, Dense, Flatten
from keras.models import Sequential

kernel_size = 3
```

```
model = Sequential()
model.add(Conv1D(filters=32, kernel_size=kernel_size, input_shape=(16, 256),
    padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.2))
model.add(Flatten())
#fully connected layer
model.add(Dense(32, input_shape=(64, )))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
model.add(Dropout(0.2))
model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

Optimizers are required for compiling a Keras model. Keras has a lot of optimizers, but the most used are the AdaDelta, SGD, Adam and RMSprop. In the scope of this dissertation, it is used Adam and RMSprop because are the ideal for classification problems. In Keras, for binary classification it is used the RMSprop, which is what is intended in this dissertation because we have one output that will be between 0 and 1.

Other argument when configuring the learning process is the loss function. This function is essential because quantifies the amount by which the prediction deviates from the actual values. The model tries to minimize this function, in order to prediction be practically equal to the true values. Lastly, *compile* method has metrics, which is a function that is used to judge the performance of the model. Typically is used the Accuracy metric to assess the model's performance, because in the new version of Keras were removed some legacy metric functions such as Matthews coefficient correlation and Recall. In this dissertation, the fact that will be used cross-validation enables to use some metrics that were removed in new Keras version.

4.3 Datasets adopted

To perform a prediction from a genetic mutation is required to have a dataset that contains values from the features, or the DNA sequence with the respective nucleotides, which can be already in codon's format. In chapter 2 were stated some datasets, but in this dissertation, will be used five: **HumVar**, **ExoVar**, **PredictSNP**, **Swissvar** and **Varibench**.

Datasets	Deleterious variants	Neutral variants	Total
Exovar	*5156*	*3694*	*8850*
Humvar	*21090*	*19299*	*40389*
PredictSNP	*10000*	*6098*	*16098*
Swissvar	*4526*	*8203*	*12729*
Varibench	*4309*	*5957*	*10266*

Table 4.1: Variants of each dataset

The table above shows the number of deleterious, neutral and total variants. Deleterious means the variants who are harmful or injurious, neutral means the opposite meaning of deleterious variants, and the total are the number of deleterious plus the neutral variants. These datasets were chosen to be used in this dissertation, because we have already available data from these five datasets. For an initial phase of this dissertation, the five datasets have feature values, which were calculated by mathematical formulas. ExoVar is the smallest dataset and HumVar is the biggest one. In these five datasets, the first column represents the gene's identifier, the position where the mutation occurs and the respective mutation performed in the sequence:

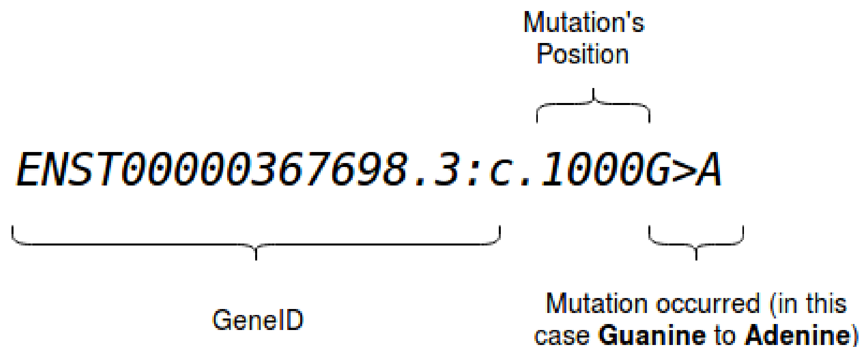


Figure 4.5: Representation of geneID, mutation's position and mutation

The second column is the class labels, i.e, -1 if the mutation is pathogenic and 1 if is non-pathogenic before training and classification. The remaining columns are represented by its decimal values, obtained with mathematical formulas. The section 4.4 mentions the steps that should be performed when using this dataset format.

In the next phase of this dissertation, the datasets content are simplified, and now we have only four columns with original sequence and mutated sequence represented with the

nucleotides (letters). Such as the previous format of the dataset approached later, the first column represents the gene's identifier, the position of the mutation and the respective mutation. The second represents the class labels (positive 1, -1 negative) and now, the third column is represented by the original DNA sequence composed by the sequence of nucleotides, in which the middle position of the sequence is the nucleotide that will suffer the mutation, such as the fourth and last column. This column contains the mutated DNA sequence, and a different nucleotide from the original sequence, which represents the mutation performed. The deployment of this task is performed in section 4.5.

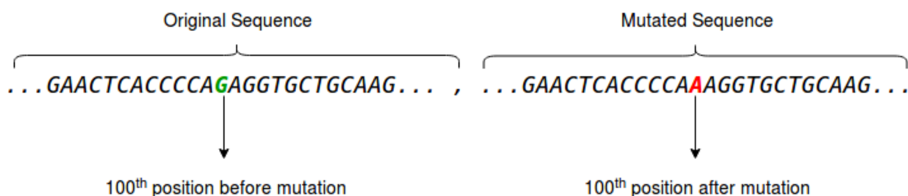


Figure 4.6: Mutation performed from Guanine (G) to Adenine (A)

Another composition of the datasets (and the last used) is similar to the second, mentioned previously. As was stated in chapter 1, DNA sequence is composed by nucleotides which groups of three nucleotides generate codons. So, this means that the DNA sequence in the dataset is now displayed in codons, as the Figure 4.7 shows below. Also, this time the position where the mutation occurs is different, because we have a window with less codons.

Figure 4.7: Mutation performed from Guanine (G) to Adenine (A) in codon sequence

4.4 Predictions using classical classifiers

As stated previously, in an initial phase of the dissertation was performed predictions with the scikit-learn's classical classifiers, in order to know the results obtained applying these classifiers in all datasets with and without cross-validation. The application of these classifiers in the datasets will help to understand how each classifier behaves, taking into account its performance and its scoring metrics such as AUC, Accuracy and MCC.

The first thing to do was to read the dataset, using the *Numpy* library. After reading the dataset it is essential to pre-process the data, in order to specify what data will be used by the classifier (class labels). Then, is specified the classifier to be used by the data for training and test, but in order to get the final score, taking into account the metric used, the model must be fitted to specify the training data and the target values (or class labels). The method that performs this operation is called *fit()*. Once the data is fitted, it is time to compute the score, based on each metric mentioned. To get the score, using accuracy or MCC as metrics, is required to predict the class labels for the provided data. The method used for this is *predict()*, which predicts if the mutation is pathogenic or not (the method uses -1 to non-pathogenic and 1 to pathogenic). Then, these predictions are used in *accuracy_score* and *matthews_corrcoef* methods to get the final score. The AUC metric receives the input data and the target score, i.e, the probability estimates of the training data. To obtain these probability estimates, is used the method *predict_proba()* on the training data, and then is possible to get the result using AUC metric with *roc_auc_score* method.

Snippet 4.11: Decision Tree with accuracy scoring metric

```
from sklearn.tree import DecisionTreeClassifier
import numpy as np
from sklearn.metrics import accuracy_score

cv = 10
dataset_file = "exovar_features.txt"

def read_and_preprocessing_data(file):
    file_data = np.genfromtxt(file, delimiter=',', skip_header=0) # delimiter ->
        because features are separated by commas
    # skip_header -> skip no line in the beginning of the file
    true_class = np.ravel(file_data[:, 1]) # get target class
    positive_fraction = (sum(true_class) * 100.0) / len(true_class)
    return file_data[:, 2:], true_class # return training data and class target

def training_and_test(inputX, inputY):
    dt = DecisionTreeClassifier(random_state=0)
    dt.fit(inputX, inputY)
    score = accuracy_score(inputY, dt.predict(inputX))
    mean = score.mean()
```

With cross-validation there is no need to specify the *fit()* and *predict()* methods, because it is possible to use the **Pipeline** module. This module has the purpose of assembling several steps that can be cross-validated together, and its intermediate steps execute fit and transform

methods. The method *make_pipeline* constructs the pipeline from the given estimators and it is a shorthand for the Pipeline constructor. Typically, this method receives a **StandardScaler**, which unify features by removing the mean and then scale to unit variance. The StandardScaler makes the centering and scaling happen on each feature of the training set. Then, mean and standard deviation are stored to be used on later data using the transform method. Standardization is a common requirement for machine learning estimators, because many elements used in the objective function of a learning method consider that all features and its variance are around 0.

Snippet 4.12: Decision Tree using cross-validation with accuracy scoring metric

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from sklearn import preprocessing
from sklearn.pipeline import make_pipeline
import numpy as np

cv = 10
dataset_file = "exovar_features.txt"

def read_and_preprocessing_data(file):
    file_data = np.genfromtxt(file, delimiter=',', skip_header=0)
    true_class = np.ravel(file_data[:, 1]) # get target class
    positive_fraction = (sum(true_class) * 100.0) / len(true_class)
    return file_data[:, 2:], true_class

def training_and_test(inputX, inputY):
    dt = DecisionTreeClassifier(random_state=0)
    mkp = make_pipeline(preprocessing.StandardScaler(), dt)
    score = cross_val_score(mkp, inputX, inputY, cv=cv, scoring='accuracy')
    mean = score.mean()
```

Lastly, it was performed classification using the cross-validation models selection: Stratified K-fold and Group K-fold. The implementation is similar for each model selection, being that the Group K-fold needs the groups, which will be the gene identifier (geneID).

Snippet 4.13: Decision Tree using Stratified K-fold model selection

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn import preprocessing
from sklearn.pipeline import make_pipeline
import numpy as np
```

```
n_splits = 10
dataset_file = "exovar_features.txt"

def read_and_preprocessing_data(file):
    file_data = np.genfromtxt(file, delimiter=',', skip_header=0)
    true_class = np.ravel(file_data[:, 1]) # get target class
    positive_fraction = (sum(true_class) * 100.0) / len(true_class)
    return file_data[:, 2:], true_class

def training_and_test(inputX, inputY):
    dt = DecisionTreeClassifier(random_state=0)
    skf = StratifiedKFold(n_splits=n_splits)
    mkp = make_pipeline(preprocessing.StandardScaler(), dt)
    score = cross_val_score(mkp, inputX, inputY, cv=skf, scoring='roc_auc')
    mean = score.mean()
```

Snippet 4.14: Decision Tree using Group K-fold model selection

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score, GroupKFold
from sklearn import preprocessing
from sklearn.pipeline import make_pipeline
import numpy as np

n_splits = 10
dataset_file = "exovar_features.txt"

def read_and_preprocessing_data(file):
    file_data = np.genfromtxt(file, delimiter=',', skip_header=0)
    true_class = np.ravel(file_data[:, 1]) # get target class
    groups = np.genfromtxt(file, delimiter=',', skip_header=0, dtype='str')
    file_data_gene = np.ravel(groups[:, 0])

    positive_fraction = (sum(true_class) * 100.0) / len(true_class)
    return file_data[:, 2:], true_class, geneID

def training_and_test(inputX, inputY):
    dt = DecisionTreeClassifier(random_state=0)
    skf = GroupKFold(n_splits=n_splits)
    mkp = make_pipeline(preprocessing.StandardScaler(), dt)
    score = cross_val_score(mkp, inputX, inputY, cv=skf, scoring='roc_auc')
    mean = score.mean()
```

4.5 Predictions in nucleotide and codon sequence

In this part of the dissertation will be present the different encoding strategies used to perform pre-processing, training and classification. It is necessary to encode the data because it is present as a nucleotide sequence (or DNA sequence) in string format, and obviously we cannot ”work” over this type of data. Another reason for encoding is the type of data used by the prediction models, which not accept data string format. Thus, taking these statements into account, is necessary to use different encoding strategies not only for training and classification but also to know what is the best encoding strategy.

So, the strategies used were **One-Hot DNA sequence**, **65 One-Hot vector encode**, **One-Hot vector amino acid encode**, **Amino acid properties vector**, **Amino acid physical-chemical properties**, **Amino acid properties based on a Venn diagram**.

4.5.1 One-Hot DNA sequence

For this encoding strategy, the data is pre-processed as a nucleotide sequence. So, each letter must have a special encoding that differentiates from the other letters. The following Figure 4.8 illustrates this:

A	C	G	T	X
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

Figure 4.8: One-Hot Vector encode for each nucleotide

Most examples performed with machine learning use a Numpy array to encode the data, because is an effective technique to use. The codification presented in Figure 4.8 was used in work [46] and it is called *One-Hot vector* encode. This is a group of bits in which all combinations are encoded in a vector with a single 1 and the remaining bits are 0.

Taking into account the codification of each nucleotide represented in Figure 4.8, then a dictionary was used in which each nucleotide is the key and the respective vector is a numpy array value, i.e, each nucleotide is replaced by the its vector. Due to the fact that the codification of the sequence is performed by One-Hot vector, this strategy could be called as *One-Hot vector encode*.

Snippet 4.15: Dictionary used to encode the DNA sequence

```
import numpy as np

dictionary = {
    'A': np.array([1, 0, 0, 0, 0]),
    'C': np.array([0, 1, 0, 0, 0]),
    'G': np.array([0, 0, 1, 0, 0]),
    'T': np.array([0, 0, 0, 1, 0]),
    'X': np.array([0, 0, 0, 0, 1])
}
```

Once we have a dictionary to replace each nucleotide, it is time to perform the reading and pre-processing of the data. This operation is done similarly to the previous one in section 4.4, but this time the dataset has the original sequence, i.e, the sequence without any mutation, and the mutated sequence which contains a single mutation. Some samples have 'X' as a nucleotide. This letter is used just to adjust the window of genetic code, i.e, for instance the mutation could be occur in the first five, ten positions of the genetic code. Using this letter, the mutations occur in the same position for all samples of the dataset. Another reason to use this letter is the emergence of sequencing errors. As the section 4.3 mentions, the dataset has 100 nucleotides before the position where the mutation occurs, and 100 after this position, which means that we have 201 nucleotides in the original and mutated sequence.

In the pre-processing of the data, is specified what is the original and mutated sequence, as well as the different techniques that be used in data to perform classification, using k-fold cross-validation. One of the techniques used was *subtract*, which consists into subtract the original sequence with the mutated sequence. Numpy library performs this operation with *subtract()* method and it is like a subtraction between two matrices, so it is not necessary to perform a reshape of the data. This technique facilitates the classification and improves the performance of the model, because only one index will have number 1, one -1, and the remaining data will have only zeros.

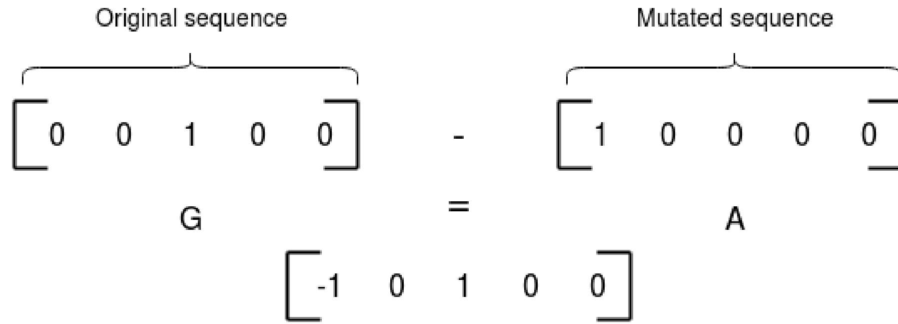


Figure 4.9: Mutation performed from Guanine to Adenine represented in vectors.

The other technique used was *vstack*, which consists into taking a sequence of arrays and stack them vertically to make a single array. In the concept of this dissertation, this function handles the mutation sequence and puts it below the original mutation, causing a change in the shape of the data which will have the double of the samples. The performance of this model with this shape of the dataset is very low, so it is essential to perform a reshape of the dataset, which can be performed by the method *reshape()*.

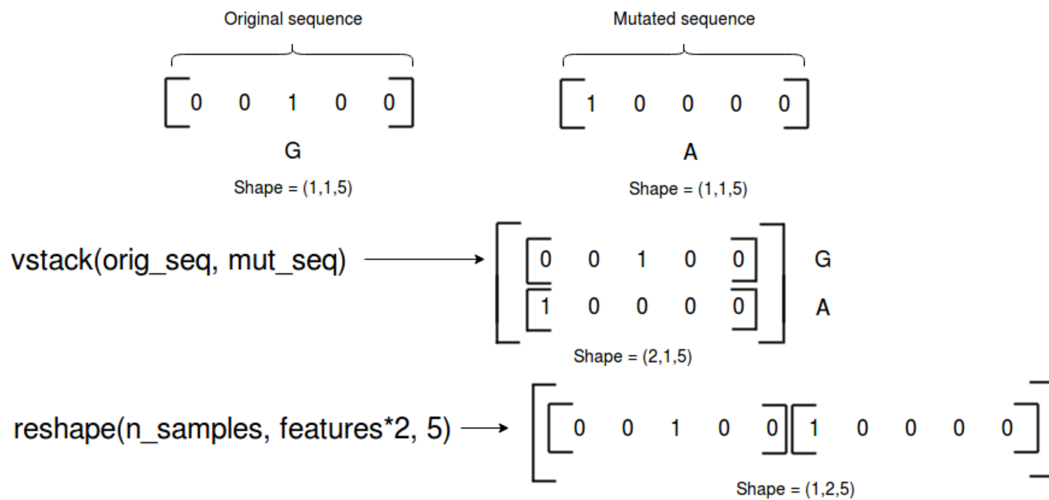


Figure 4.10: Example of reshape and vstack with the respective sequences.

After pre-processing the data with the strategies approached, it is performed the training and classification. With the datasets in a format of nucleotide sequences, the models used to perform training and classification were the Convolutional Neural Network (approached in chapter 2 and studied in work [37]) and Complex Network models, which will be explained further ahead in section 4.6.

4.5.2 65 One-Hot vector encode

In this strategy, the data is in codon format. So, it is assigned to each codon a one-hot vector with a length corresponding to the number of existing codons in the datasets, i.e, we have 64 codons and the codon "XXX". These vectors are different from each other. This strategy is similar to the one who was developed previously, as the Figure 4.11 shows.

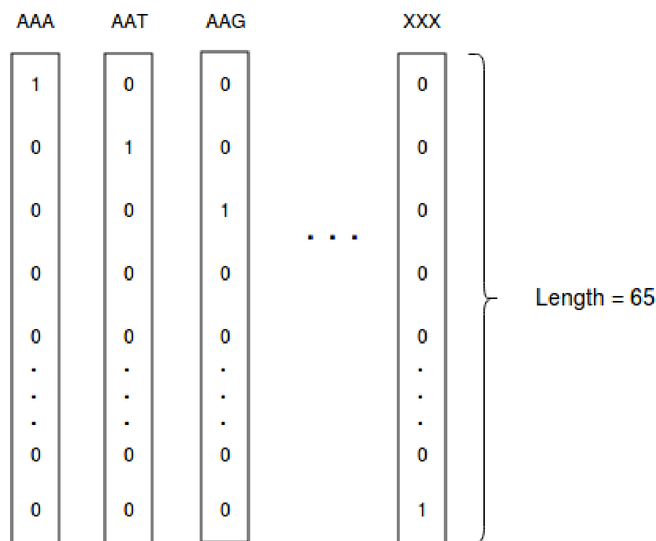


Figure 4.11: 65 One-Hot Vector encode for each codon.

As stated in section 4.3, we have 30 codons before and after the codon mutated. This means that we have 61 codons. The pre-processing is performed as in section 4.5 being able to modify the data window, i.e, instead we have 30 codons before and after the mutated codon, we can have less than 30 codons, specifying the amount desired from the mutated position. Performing this modification reduces the amount of data to analyze and improves the final performance of the model. Over the five datasets with this format, it was applied *subtract* and *vstack* operations between original and mutated sequence. Applying *subtract* operation, the first Convolutional layer of the model will have a *input_shape* variable of (61, 65), because we still have 61 codons in the sequences and each codon vector have a length of 65, as shown in Figure 4.11. Using *vstack* operation, *input_shape* variable will be equal to (122, 65) because, as shown in Figure 4.10, we have to perform the *reshape* operation to keep the number of

dataset samples.

Classification is performed such as the last chapter, using the same model and its parameters, differentiating only the shape of input. It is used the KerasClassifier with the same epochs and batch size, pipeline module to perform predict and fit with KerasClassifier and are also used the two k-fold cross-validation variants *StratifiedKFold* and *GroupKFold*.

4.5.3 One-Hot vector amino acid encode

This strategy consists into assign an One-Hot vector to a codon, according to the resulting amino acid, i.e, the codons which encode the same amino acid have the same vector value, as shown in Figure 4.13. As stated in chapter 1, the amino acids result during protein synthesis. In order to understand how codification process will be performed, it is essential to present in first all amino acids and their abbreviations that will be used in this dissertation. This information is shown in Figure 4.12[1] helps to know what is the resulting amino acid, and which codons encode the same amino acid.

One letter code	Three letter code	Amino acid	Possible codons
A	Ala	Alanine	GCA, GCC, GCG, GCT
B	Asx	Asparagine or Aspartic acid	AAC, AAT, GAC, GAT
C	Cys	Cysteine	TGC, TGT
D	Asp	Aspartic acid	GAC, GAT
E	Glu	Glutamic acid	GAA, GAG
F	Phe	Phenylalanine	TTC, TTT
G	Gly	Glycine	GGA, GGC, GGG, GGT
H	His	Histidine	CAC, CAT
I	Ile	Isoleucine	ATA, ATC, ATT
K	Lys	Lysine	AAA, AAG
L	Leu	Leucine	CTA, CTC, CTG, CTT, TTA, TTG
M	Met	Methionine	ATG
N	Asn	Asparagine	AAC, AAT
P	Pro	Proline	CCA, CCC, CCG, CCT
Q	Gln	Glutamine	CAA, CAG
R	Arg	Arginine	AGA, AGG, CGA, CGC, CGG, CGT
S	Ser	Serine	AGC, AGT, TCA, TCC, TCG, TCT
T	Thr	Threonine	ACA, ACC, ACG, ACT
V	Val	Valine	GTA, GTC, GTG, GTT
W	Trp	Tryptophan	TGG
X	X	any codon	NNN
Y	Tyr	Tyrosine	TAC, TAT
Z	Glx	Glutamine or Glutamic acid	CAA, CAG, GAA, GAG
*	*	stop codon	TAA, TAG, TGA

Figure 4.12: Codons and amino acids

[1] http://www.hgvs.org/mutnomen/codon.html

As we can see in the table above, for instance the codons ”TTT” and ”TTC” result in the same amino acid called Phenylalanine (phe). ”CCT”, ”CCC”, ”CCA” and ”CCG” result in the same amino acid called Proline, and other many examples. According to the codon table, we have a total of 20 amino acids, 3 codons are STOP signals, and the codon ”XXX” is considered to result in an another amino acid. So, taking into account what was said, the codons in the sequence will be encoded using One-Hot vector, but this time the length of the vectors will be 22 due to the fact that we have 20 amino acids, STOP codons and a different amino acid from codon ”XXX”.

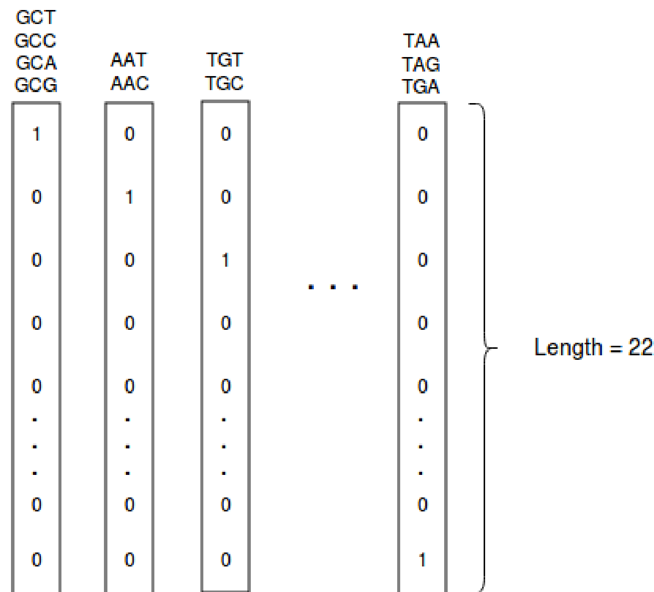


Figure 4.13: One-Hot vector encode for amino acids

This time, the shape of input from the first Convolutional layer will be equal to (61, 22) for *subtract* and (122, 22) for *vstack* followed by *reshape* operation. It is used the same model, with the same parameters, so the classification process is executed in the same way using the two k-fold cross-validation variants Stratified K-fold and Group K-fold.

4.5.4 Amino acid properties as a vector with ones and zeros

This encode strategy is based on a list of amino acid properties. Each codon is replaced by a vector (*numpy.array*) with zeros and ones, which are assigned according to the table[2] present below.

Property	Amino Acids
Acidic / amide	*Asp, Glu, Asn, Gln*
Charged negative	*Asp, Glu*
Charged positive	*Lys, Arg*
Polar	*Ala, Gly, Ser, Thr, Pro*
Hydrophobic	*Val, Leu, Ile, Met*
Size big	*Glu, Gln, His, Ile, Lys, Leu, Met, Phe, Trp, Tyr*
Size small	*Ala, Asn, Asp, Cys, Gly, Pro, Ser, Thr, Val*
Aliphatic	*Ile, Leu, Val*
Aromatic	*His, Phe, Tyr, Trp*

Table 4.2: Properties of amino acids

Taking into account the table above, we have 9 properties and the amino acids which owns these properties. As we can see, an amino acid can have one or more properties, so this means that the encoding of the data is similar to One-Hot vector. The Figure 4.14 explains in detail how the encoding process is performed.

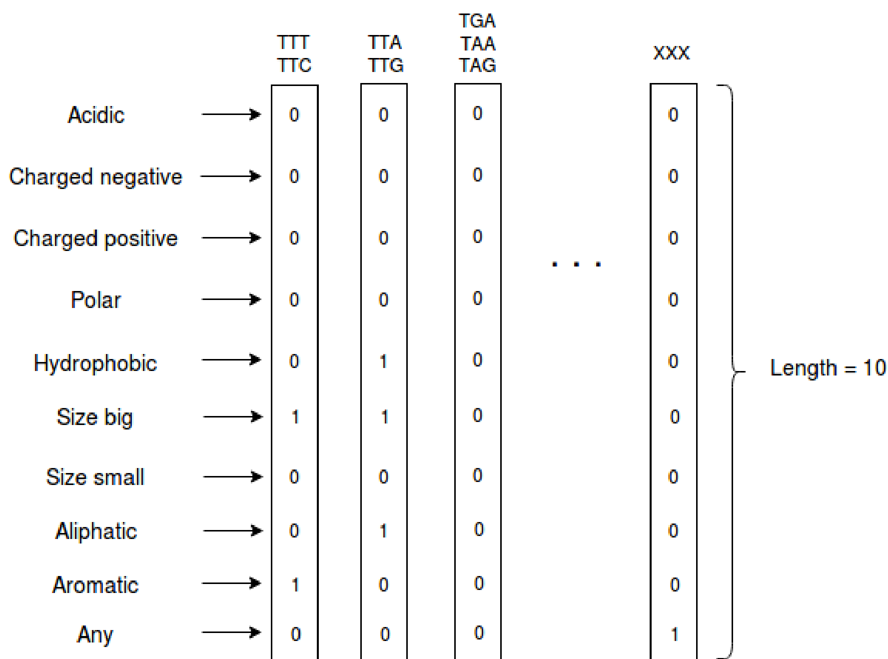


Figure 4.14: Encoding with properties of amino acid

[2]http://www.hgvs.org/mutnomen/codon.html#aaprops

In Figure 4.14, the codons are replaced by the respective vectors, which are made according to the properties of the resulting amino acids. Codons "TTT" and "TTC" result in amino acid Phenylalanine which is an aromatic and big size amino acid. The STOP signals are replaced by vectors with all indexes equal to 0, because they do not contain any of the properties presented. The extra codon "XXX" has the encoding that is in the figure because it was decided to add an index only for this codon in order to differentiate from the other codons.

After this data pre-processing, the training of the model is executed with the shape of input equal to (61, 10) for the first Convolutional layer using *subtract* operation, and are used *vstack* and *reshape* operations when the shape of input is equal to (122, 10). It is used the same model from the previous sections and the classification is performed using the two k-fold cross-validation variants.

4.5.5 Venn diagram with amino acid properties

Similarly to the last strategy approached, in this section the codons are encoded in vectors too. These vectors have ones and zeros too, but these are assigned according to the Venn's diagram, displayed in Figure 4.15[3], which is based on a work present in the footnote. The amino acid properties are the same as the previous strategy with the difference that here is present one more amino acid property.

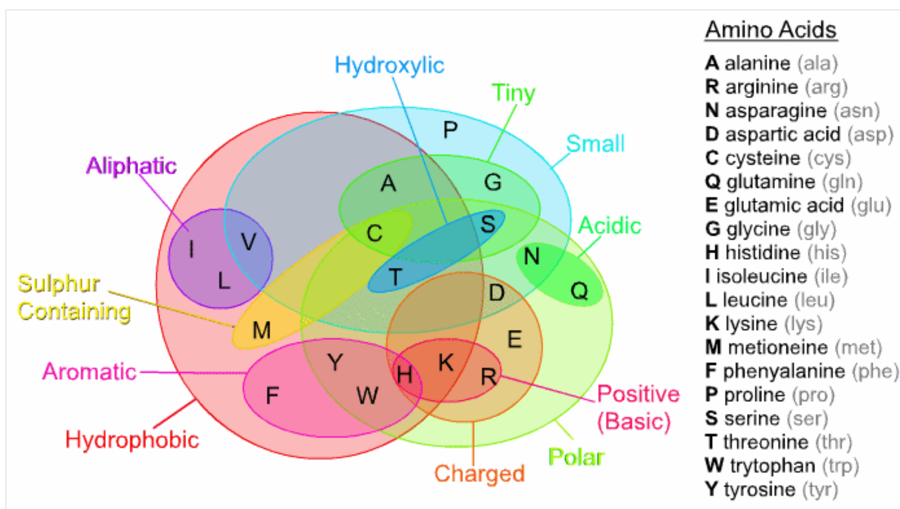


Figure 4.15: Venn's diagram for properties of amino acids.

In pre-processing data, the codons are encoded according the figure above. For example, the amino acid Alanine has "Tiny", "Small" and "Hydrophobic" properties because it is within the 3 circles, Lysine (represented by letter "K") has "Positive (Basic)", "Charged", "Polar" and "Hydrophobic" properties. Here, we have 11 properties, which means that the encoding

[3]https://amit1b.wordpress.com/the-molecules-of-life/about/amino-acids/

vectors will have a length of 11, and the respective encoding is performed similarly to the previous sections, which is shown in Figure 4.16 below.

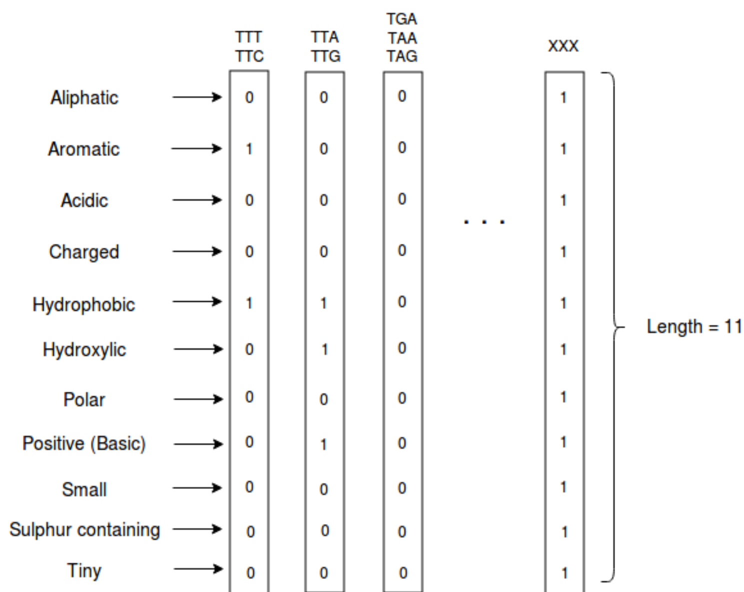


Figure 4.16: Encoding according to Venn's diagram.

The codon "XXX" is encoded in vector which all indexes are 1, because it is a way to differentiate from the other codons (the same reason mentioned in the previous sections) and the STOP signals have their all indexes equal to zero. Once the encoding is now finished, we can train the model (the same as the previous sections) with the shape of input equal to (61, 11) for the first convolutional layer using *subtract* operation, and equal to (122, 11) using *vstack* and *reshape* operations. Then it is performed the classification using the two K-fold cross-validation variants.

4.5.6 Amino acid physical-chemical properties

Another technique similar to the previous is approached in this section, but instead to encode the codons into the properties of the resulting amino acids as vectors with zeros and ones, the data is encoded using the physical-chemical properties of amino acids, which are based on work [47], but the figure is retrieved from article [33].

	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
hydropathy	1,8	2,5	-3,5	-3,5	2,8	-0,4	-3,2	4,5	-3,9	3,8	1,9	-3,5	-1,6	-3,5	-4,5	-0,8	-0,7	4,2	-0,9	-1,3
polarity	8,1	5,5	13,0	10,5	5,2	9,0	10,4	5,2	11,3	4,9	5,7	11,6	8,0	12,3	10,5	9,2	8,6	5,9	5,4	6,2
averageMass	89,1	121,2	133,1	147,1	165,2	75,1	155,2	131,2	146,2	131,2	149,2	132,1	115,1	146,1	174,2	105,1	119,1	117,1	204,2	181,2
isoElectricPt	6,0	5,1	2,9	3,2	5,5	6,1	7,6	6,1	9,6	6,0	5,7	5,4	6,3	5,7	10,8	5,7	5,6	6,0	5,9	5,6
vanDerWaals	67,0	86,0	91,0	109,0	135,0	48,0	118,0	124,0	135,0	124,0	124,0	96,0	90,0	114,0	148,0	73,0	93,0	105,0	163,0	141,0
netCharge	0,0	0,0	0,0	0,0	0,0	0,2	0,0	0,0	0,0	0,1	0,0	0,0	0,2	0,0	0,0	0,0	0,0	0,1	0,0	0,0
pK1	2,4	1,9	2,0	2,1	2,2	2,4	1,8	2,3	2,2	2,3	2,1	2,1	2,0	2,2	1,8	2,2	2,1	2,4	2,5	2,2
pK2	9,9	10,7	9,9	9,5	9,3	9,8	9,3	9,8	9,1	9,7	9,3	8,7	10,6	9,1	9,0	9,2	0,0	9,7	9,4	9,2
pKa	0,0	8,2	3,9	4,1	0,0	0,0	6,0	0,0	10,5	0,0	0,0	5,4	0,0	0,0	12,5	5,7	5,5	0,0	5,9	10,5

Figure 4.17: Amino acid physical-chemical properties [47]

The letters present as columns in the table above represent each amino acid (see Figure 4.12), which are pre-processed into a dictionary that encode the data. For the STOP signals are given a random value for its properties and it is performed the same for the codon ”XXX”.

In the pre-processing data, the original and mutated sequence are separated from each other into numpy arrays, as in other techniques. Then, it is only performed a subtract operation between each sequence array in order to simplify the training and classification, and it is the best method to get better results (see chapter 5). Some of these values are used in the initial phase of this dissertation when is performed the training and test of each classical classifier on the datasets, and these values are also obtained by mathematical formulas from another works.

4.6 Models developed

In order to perform training and classification, it is necessary to build some classifiers/-models. In this section are explained in detail all procedures performed in the buiding of these models. One of them is a CNN and the other is a Complex Network.

4.6.1 Convolutional Neural Network

As stated previously, CNN is a sequential model. So, this model has Convolutional layers, Dropout layers, Max Pooling layers and then a FCNN with Dense (hidden) layers and neurons per layer. The beginning of this implementation starts to create a sequential model. Then, are added the convolutional layers specifying the parameters such as filters, the size of the kernel, the shape of the input, activation function and number of strides (the respective role of each parameter is explained in subsection 4.2.3). After each convolution layer, it was then added a single MaxPooling layer and a Dropout with a rate of 0.2 (20%), in order to reduce the amount of parameters and overfitting problem. In general were added two convolutional layers, as well as two MaxPooling layers and two Dropout (both with a rate of 0.2), because it helps to improve the final score and train the model better.

After adding the Convolutional, MaxPooling and Dropout layers, we can add the Fully Connected layers to complete the model. In order to achieve this, is essential to add a Flatten layer to perform the connection between the Convolutional, MaxPooling and Dropout layers and the Fully Connected layers. Fully Connected layers compose a FCNN, which may contain Dropout layers between each layer, such as happen in Convolutional Neural Networks. In this case, was added 3 Fully Connected layers with each one has 100 neurons and a $ReLU$ activation function, 3 Dropout layers with a rate of 0.2 and one Output layer with 1 neuron and a sigmoid as activation function, because it is pretended to have an output between 0 and 1. The reason to have this amount of layers and the values for each parameter is due to the fact that we can get better results, taking also into account the performance of each model, i.e, the time of pre-processing data and time of training and classification. So, the chosen model and its parameters is like an optimization between the final result (score) and the performance of the model. It is important to refer that in *model.compile()*, the loss function is equal to ”binary_crossentropy”, because we pretend to perform a binary classification, i.e, classify if a mutation is pathogenic or not (values 0 for non-pathogenic and 1 for pathogenic), and the optimizer used is the rmsProp because is recommended by Keras to perform binary classification. The snippet below shows the deployment of the model. Each parameter is explained in subsection 4.2.3 and the other values attributed are the ones who can get better results and most used in some other works.

Snippet 4.16: Implementation of Convolutional Neural Network model

```
from keras.layers import Dense, Conv1D, MaxPooling1D, Dropout, Flatten
from keras.models import Sequential

filters = 32
kernel_size = 3
n_chars = 201
num_of_nucleotides = 5
num_strides = 1

def baseLineModel():
    model = Sequential()
    model.add(Conv1D(filters=filters, kernel_size=kernel_size,
        input_shape=(n_chars, num_of_nucleotides), padding='same',
        activation='relu', strides=num_strides))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Dropout(0.2))
    model.add(Conv1D(filters=filters, kernel_size=kernel_size, padding='same',
        activation='relu'))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(100, kernel_initializer='normal', activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(100, kernel_initializer='normal', activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(100, kernel_initializer='normal', activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='rmsprop',
        metrics=['accuracy'])
    return model
```

Once the model is built, it is time to perform training and classification. As we pretend to use k-fold cross-validation and Scikit-learn API to achieve the goal, it was used the Keras wrapper *KerasClassifier*. As stated in subsection 4.2.3, this wrapper has a parameter that builds, compiles and returns a Keras model, so, taking into account the snippet above we have the following statement: *KerasClassifier(build_fn = baseLineModel)*. In this dissertation, the epochs was assigned the value 15, and for batch size 128 for this model, because when tests were carried out, these values were the ones who can get the best results in terms of performance and score. Then, it was applied the Pipeline to the KerasClassifier and used

Group-K fold and Stratified-K fold cross-validation to get the final classification score. The snippet below considers the model built in the last snippet.

Snippet 4.17: Classification of the model using k-fold cross-validation

```
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score, StratifiedKFold, GroupKFold
from sklearn.pipeline import Pipeline

estimators = [('cnn',KerasClassifier(build_fn=baseLineModel, epochs=15,
    batch_size=128, verbose=0))]
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, random_state=0)
results = cross_val_score(pipeline, sequence, class_y, cv=kfold, scoring='accuracy')
```

Below is shown the respective model described in these snippets:

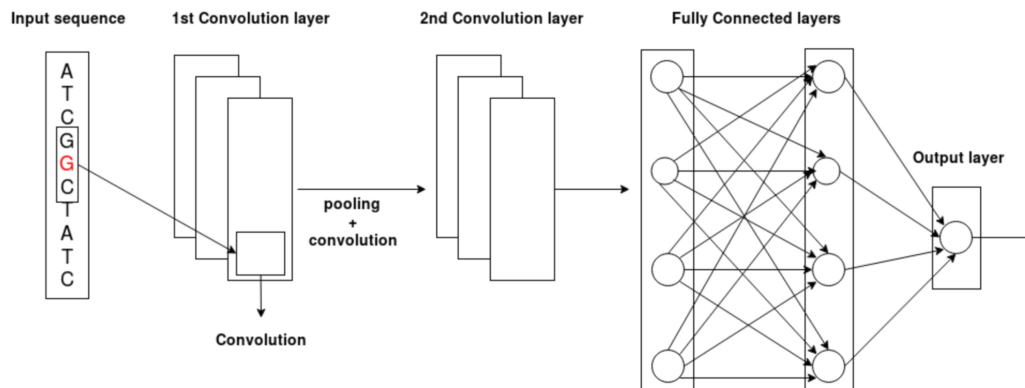


Figure 4.18: Convolutional Neural Network model [37]

4.6.2 Complex Network

In this section, the model used to be trained is not a sequential model but is a Complex Network model, which is based on the Network referenced in work [37]. This network is composed by 2 inputs, 2 Convolutional Neural Network, a Neural Network and 1 output. One of the Convolutional Neural Network receives as input the original sequence and the other receives the mutated sequence. Then, the respective outputs are merged as inputs of the Neural Network (MLP), which in the last layer contains an output that gives a value score between 0 and 1.

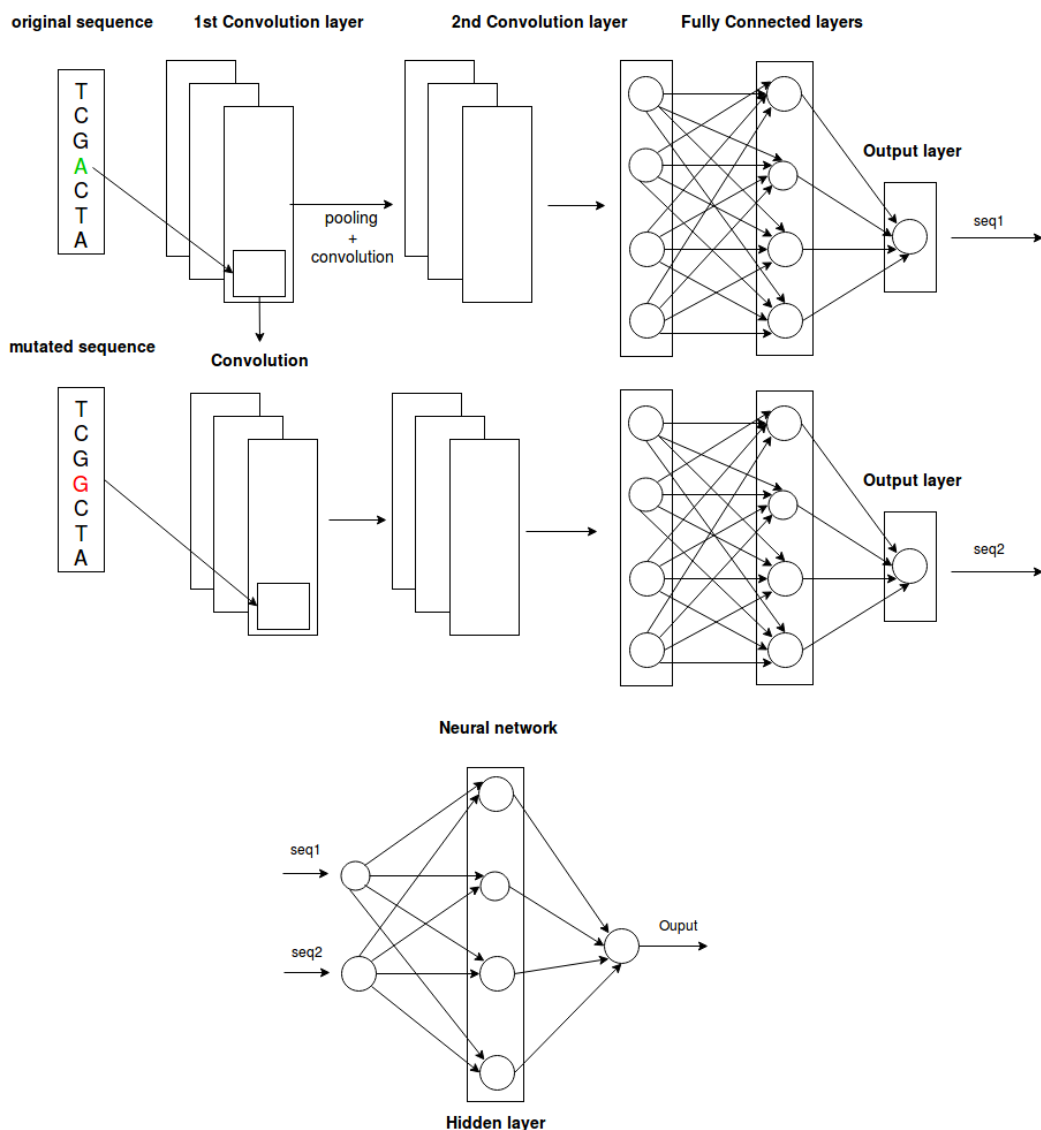


Figure 4.19: Complex Network model [37]

As Figure 4.19 shows, the output of each Convolutional Network is an input of the Neural

Network. The creation of this model is similar to the previous ones. It starts by creating two sequential models individually with the same parameter values as the previous models and then, instead of to compile the two models, their outputs are concatenated in order to make them as inputs of the Neural Network. The result of the concatenation is then connected to the first Neural Network layer which connects with its hidden layers and then to the final output layer. The diagram represented in Figure 4.20 explains better the steps that were performed to build this model.

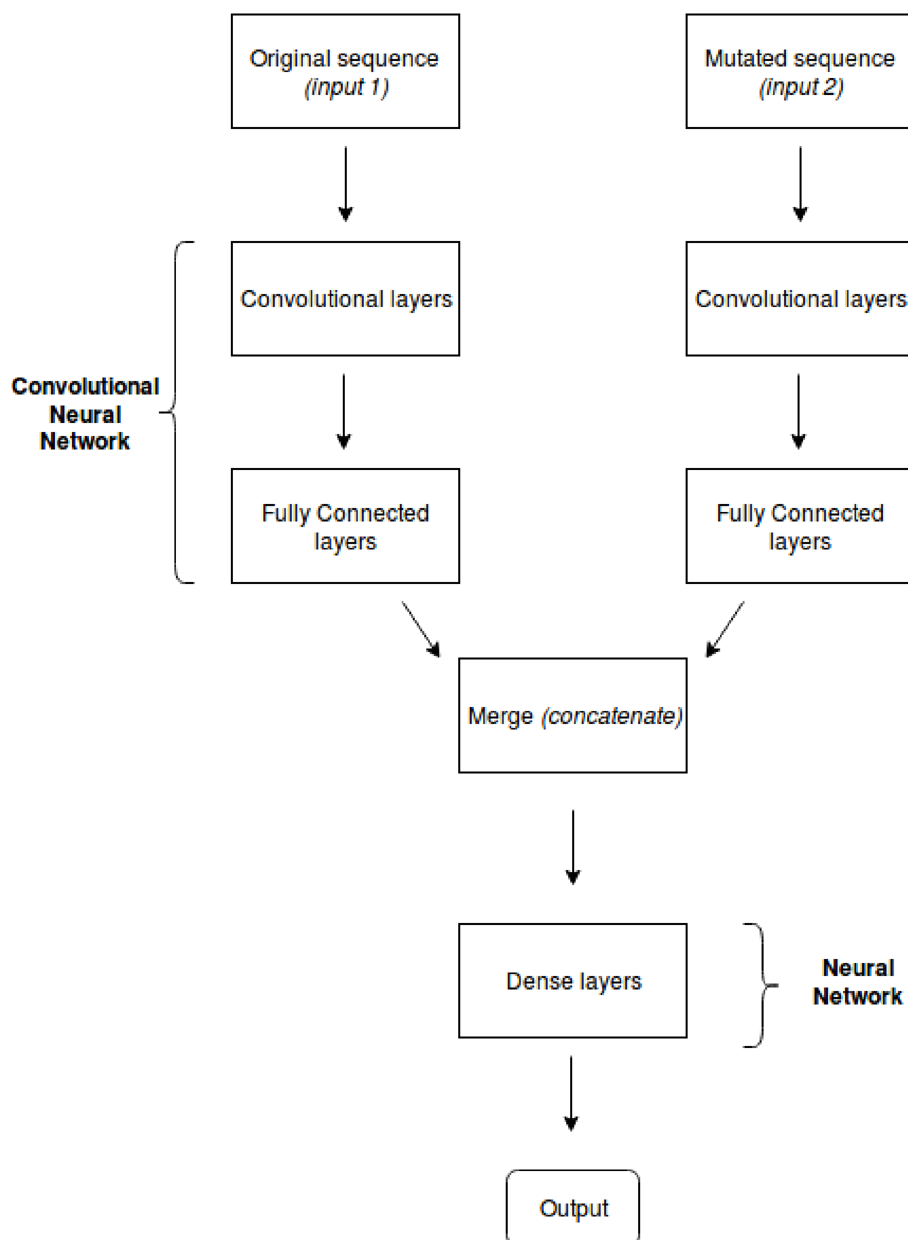


Figure 4.20: Diagram of the Complex Network model

The model is compiled in the same way as the models of previous sections, i.e, uses the loss function *binary_crossentropy*, and the optimizer *rmsProp*. Then, it is used the method *fit()* in order to the estimator learn from the model, with 20 epochs and a batch size equal to 32. The reason to use these values is due to the fact that it obtains (in average) better score. The performance of the model and the final result will be discussed in chapter 5. Below are the code snippets of each block of the diagram and its explanation.

Snippet 4.18: Convolutional Neural Network model

```
from keras.layers import Dense, Conv1D, MaxPooling1D, Dropout, Flatten
from keras.models import Sequential

num_codons = 61
vector_length = 22 # used amino acid property codification

#the same is performed for mutated sequence (model2)
model1 = Sequential()
model1.add(Conv1D(filters=32, kernel_size=3, input_shape=(num_codons,
    vector_length), padding='same', activation='relu'))
model1.add(MaxPooling1D(pool_size=2))
model1.add(Dropout(0.2))
model1.add(Conv1D(filters=filters, kernel_size=kernel_size, padding='same',
    activation='relu'))
model1.add(MaxPooling1D(pool_size=2))
model1.add(Dropout(0.2))
model1.add(Flatten())
model1.add(Dense(100, kernel_initializer='uniform', activation='relu'))
model1.add(Dropout(0.2))
model1.add(Dense(100, kernel_initializer='uniform', activation='relu'))
model1.add(Dropout(0.2))
model1.add(Dense(100, kernel_initializer='uniform', activation='relu'))
```

Snippet 4.19: Input layer for original sequence

```
from keras.layers import Dense, Conv1D, MaxPooling1D, Dropout, Flatten
from keras.models import Sequential
from keras.engine import Input

num_codons = 61
vector_length = 22 # used amino acid property codification
# the same is performed for mutated sequence
input1 = Input(shape=(num_codons, vector_length))
seq1 = model1(input1)
```

This snippet means that the Input engine has the dimensions of (61,22), that is, accepts 2D arrays with 61 elements (number of codons) with a length of 22 (number of digits used in encoding). The variable *seq1* is defined as one of inputs of the Convolutional Neural Network.

Snippet 4.20: Concatenation of the sequences

```
from keras.layers import Dense, Conv1D, MaxPooling1D, Dropout, Flatten, concatenate
from keras.models import Sequential
from keras.engine import Input

merged = concatenate([seq1, seq2])
```

Considering that we have already implemented the two Convolutional Neural Networks, it is time to merge the outputs of each Network, using the method *concatenate()*.

Snippet 4.21: Implementation of Neural Network

```
from keras.layers import Dense, Conv1D, MaxPooling1D, Dropout, Flatten, concatenate
from keras.models import Sequential
from keras.engine import Input, Model

merged = concatenate([seq1, seq2])
x = Dense(2, activation='relu')(merged)
x = Dense(32, activation='relu')(x)
x = Dense(1, activation='sigmoid')(x)
model = Model(inputs=[input1, input2], output=x)
model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

After deploying the Merge layer, there are gathered all conditions to deploy the Neural Network using the Dense layers. The code of the first Dense layer, which have ”(merged)” in the end of the line, means that the concatenation connects to the first layer, that is, the Neural Network receives its respective inputs. The last Dense layer is the output of the model with ”sigmoid” as activation function (explained in subsection 4.2.3) and the engine ”Model” specifies the inputs of the model and the respective output. As we can see, the inputs are seq1 and seq2 such as mentioned in the snippets below. The output is the variable ”x” because the last block of the model is the Neural Network, which is represented by the variable ”x”.

The model is now completed but still was not specified which sequence data is trained on one and the other Convolutional Neural Network. This specification is performed by the method *.fit()* which contains as arguments input ”[orig_seq, mut_seq]”. This means that the original sequence is trained in the first Convolution Neural Network and the mutated sequence is trained in the other, because the inputs of engine Model are equal to [input1, input2]. Thus, the original sequence is the input1 and the mutated sequence is the input2.

Chapter 5

Discussion and Results

In this chapter, it is performed a discussion and an analysis of the results obtained from the solutions implemented in chapter 4. It begins to analyze the results obtained from the Scikit-learn classical classifiers with k-fold cross-validation only for all datasets, such as performed in the initial phase of this dissertation. The measuring of this results is taking into account the 3 scoring metrics: AUC, MCC and Accuracy. These results are then compared to each other.

Then, it is shown the results of the predictions performed in all five datasets, which this time are coded as a DNA sequence, that is, the letters of each nucleotide. After these results, are shown the tables with the results obtained from different prediction strategies applied with the two k-fold cross-validation variants using the datasets. The performance obtained is also measured and compared between them, and as well as was performed in the initial phase of the dissertation, the final results are measured taking into account the 3 scoring metrics. Before moving forward to the tests, it is essential to know the machine hardware specifications where all models were run and tested. The models were run and tested in a machine with a CPU Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz and a Memory of 8GB.

5.1 Classical classifiers results

In this section is shown the graphics with the results obtained from the Scikit-learn classical classifiers (in the first phase of the dissertation), using 10-fold cross-validation and taking into account the three scoring metrics. These classifiers are applied on the initial datasets, those who its features are represented as numerical format. The performance for each classifier and the respective results are shown in the graphics. Finally, a comparison is then made between the scoring results and the respective results.

Each classifier uses Group k-fold and Stratified k-fold cross-validation techniques, but only the results for Group k-fold are shown. In Group k-fold, a group cannot be at the same time in training and testing set, which makes the probability of the data becoming less biased than when using Stratified k-fold cross-validation. So, this is why Group k-fold is the chosen one.

Before the analysis of the results, is important to know in what conditions they have been obtained, i.e., the values of the arguments used by each classifier. **AdaBoost** classifier uses 100 estimators due to the fact that the results are better with this value. For **Decision Tree** classifier, the score results may differ according to certain variables, mainly the minimum number of samples per leaf. The number of splits is equal to 5 only for **KNeighbors** classifier, because 10-fold cross-validation takes so long time. For the remaining classifiers, the variable is equal to 10. Another reason to use this number of folds is due to the fact that we have 10 neighbors, which will take more time to train and test the model. The weights are "uniform" because the results are better when all points in each neighborhood are weighted equally. For **Logistic Regression**, the default values were maintained. If is pretended to improve the performance, we can use all cores available. As mentioned previously in chapter 4, **Naive Bayes** uses the Gaussian Naive Bayes, because performs better for binary classification. The results, for **Neural Network** (using MLP), were obtained using 10 hidden layers, which each one has 300 neurons. **Random Forest** classifier uses the default value for all parameters. So, this means that the trees do not have a maximum depth, i.e., the number of features to consider when looking for the best split is equal to the number of samples of the dataset, and the forest contains 10 trees. Finally, for **SVM** classifier, the tests were performed only using the a kernel used in this classifier is the "linear", because tends to perform very well when the dataset is large.

So, below is shown the table with the respective results for Group k-fold cross-validation:

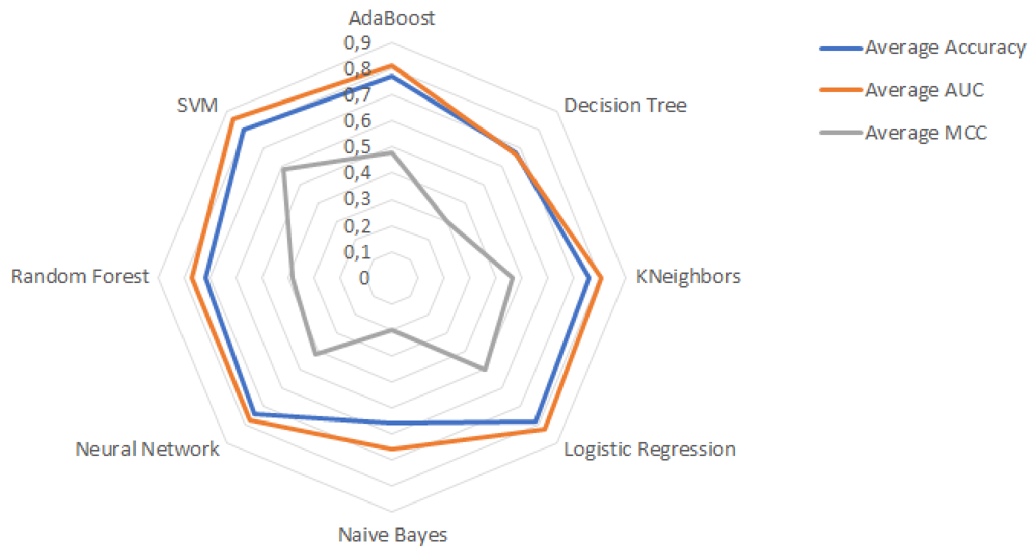


Figure 5.1: Average on dataset of Group k-fold score

The Figure 5.1 above shows the average scores on all datasets. As we can see, Logistic regression is one of the best classifiers. In AdaBoost classifier, the result of each scoring metric is nearly the same for all datasets as well as in DecisionTree classifier, except for Swissvar and Varibench dataset. For KNeighbors classifier, the analysis of the results is the same that has been made for the AdaBoost classifier, because the results are nearly the same for both datasets. If we increase the number of neighbors, the results improve, but only until a defined number of neighbors (nearly 35 or 40 neighbors on dataset). This happens because if we have few neighbors, the overfitting problem may be a reality, and if we have a lot of neighbors the data may be not trained well. Relatively to Naive Bayes, there are great differences for all scoring metrics on between all datasets. The results for NN are similar between the scoring metrics Accuracy and AUC on PredictSNP, Swissvar and Varibench datasets. The results of this classifier may variate according to the number of hidden layers and neurons per layer. If we have a lot of layers but few neurons per layer, than the data may become not well trained and then the final result decreases. On the other hand, if we have few layers and a lot of neurons per layer, we may originate the overfitting problem. Random Forest classifier results have also great differences for Accuracy and MCC metric between the datasets, but the results for AUC are almost the same for Exovar, Humvar and PredictSNP. The number of trees influences the final result, i.e., the increase of this number make the results better, but increases the training and classification time. On the other hand, the decrease of this number obtains lower results but may decreases the traning and classification time. SVM classifier is the one that gets the best results for AUC metric on all datasets. As many articles stated (one of them is [48], which compares SVM with MLP), SVM is one of the best classifiers and

probably the most used. The reason of this classifier gets great results is due to its kernel and it is more robust.

In the table below is present the comparison of these AUC scoring metric results with those obtained on [3] work, taking into account the Figure 2.3. The comparison is between AUC because is the metric with the best results:

		State of the art	
Dataset	*Best AUC*	*Best AUC*	*Non-biased*
Exovar	*0,87*	*0,92*	*0,83*
Humvar	*0,89*	*0,93*	*0,88*
PredictSNP	*0,89*	*0,93*	*0,80*
Swissvar	*0,75*	*0,73*	*0,73*
Varibench	*0,90*	*0,94*	*0,70*

Table 5.1: Comparison of AUC with the State of the Art

The table shows that the best result is obtained on Varibench dataset for AUC scoring metric. The results on Swissvar dataset are far below those obtained in the remaining datasets, such as Figure 2.2 and Figure 2.3 show us. The "Non-biased" results are obtained without the bias types mentioned in chapter 2.

Once performed the analysis about scoring metric results, then is performed the same analysis, but this time for training and classification time. All initials conditions of the classifiers are maintained, i.e., the values of the respective arguments are the same. The following graphics are represented in logarithmic scale in order to observe better the training and classification time, and also they are sorted by training time. It is important to note that the training and classification time are obtained by splitting the data into 50% for training and 50% for testing and each split runs 10 times.

Figure 5.2: Training and classification time for classical classifiers (average for 10 folds). Pre-processing time is between 2 and 11 seconds

The results of performance presented in Figure 5.2 shows that Naive Bayes is the fastest classifier. The time of pre-processing is nearly the same among each other. Naive Bayes is a fast classifier, because all it needs are prior probability values, calculated with basic operations, and can be stored ahead of time. Then, same values calculated are reused in order to calculate the posterior. Obviously that for all classifiers, the execution time is better when using the Exovar dataset, because has less samples than the others. As stated previously, Group-k fold cross-validation takes more time to perform than the Stratified k-fold cross-validation, because Group k-fold needs to collect the groups of genes available to perform its operation.

KNeighbors classifier takes a long time in classification because uses 10 nearest neighbors on each query point. In fact, this number of nearest neighbors contributes greatly to the performance of the mode. The higher the number of neighbors, the longer the execution time. Logistic Regression is also one of the fastest classifiers because is a simple classifier with low variance. NN (MLP) also takes a long time to perform. With the number of hidden layers and neurons per layer defined previously, the training and classification takes a little time. If the number of hidden layers and neurons per layer increase, the time of training and classification increases too. But the increase of hidden layers and neurons can make the score better. For instance it was tested this classifier with the same number of hidden layers, but instead to have 300 neurons per layer, we have 100 neurons per layer. The tests showed that with these values, the score decreased nearly by 1%. The training and classification time of Random Forest classifier depends on the number of trees, as well as the number of maximum features per tree and the maximum depth of trees. Increasing the number of trees, maximum features and maximum depth, the model's execution time increases too, but the score may be better. SVM classifier solves an optimization problem of quadratic order, so it takes a long time to execute with large datasets. Actually, we can say that the performance presented in

the table for Humvar is greater than the values presented. This is worst classifier in terms of performance using the conditions specified previously.

5.2 Nucleotide and codon sequence results

As well as in the previous section, in this section is also shown the tables with the results obtained from applying different techniques over the datasets, which contain a DNA sequence format. In first is shown the results from DNA sequence without the division of the datasets by codons (One-Hot DNA sequence). Each letter (nucleotide) of the dataset is represented by One-Hot vector, which has its own vector. Then are shown the results of the datasets composed by codons, using different encoding strategies. In this section, pre-processing, training and classification are performed with 15 epochs, 10 splits, a batch size of 32, a kernel size of 3 and 32 filters (see subsection 4.2.3 for more details about these variables). In this section, all strategies are used by NNs and one Complex network. Only for Complex Network it is used 20 epochs and a batch size equal to 32, because the results are better with these values. Also, for this model has been obtained the results on all datasets but without cross-validation, as well as performed in [37]. Then, the model has been tested with all strategies of encoding used.

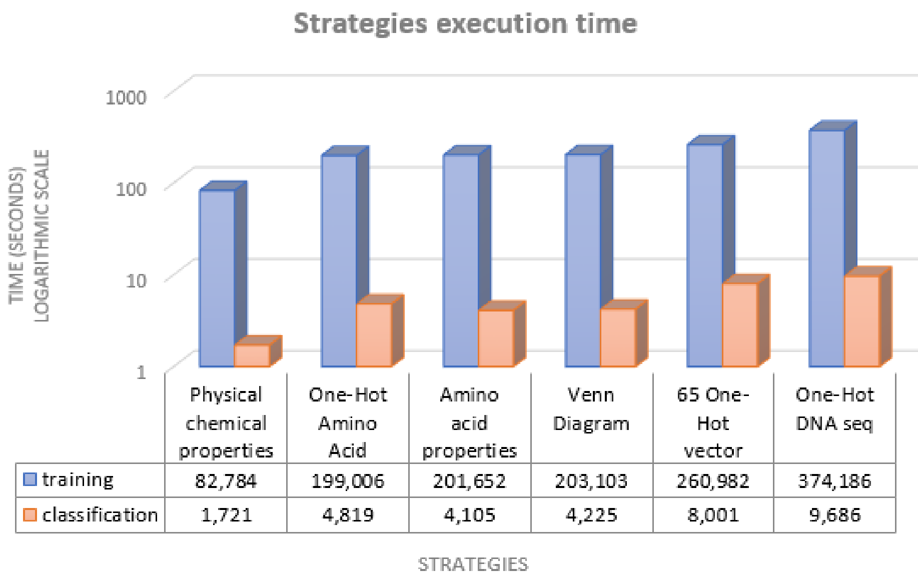
The remaining strategies are based on the codons, present in the datasets. In the first strategy, each codon is encoded with a **65 One-Hot vector encode**. So, all vectors are different from each other. The second strategy is called **One-Hot vector amino acid encode** and is similar to the previous one, but contains only 20 different vectors due to the number of amino acids that exist, one vector for Stop Signals and the other for the codon ”XXX”. The third strategy is called **Amino Acid properties** and uses vectors with a length equal to 10, because each vector index corresponds to a property of amino acid (these properties are stated in chapter 4). Then, the fourth strategy also used amino acid properties, but this time is based on a **Venn’s diagram** properties distribution. In this strategy almost all properties are the same and has one more property than the last mentioned strategy, which makes the usage of 11-length vectors. The strategy **Amino acid physical-chemical properties**, as the name states, use physical-chemical properties with the respective values in decimal format, as well as used by classical classifiers.

Lastly, has been used a different model. Instead to use only a CNN, the strategies mentioned run on a Complex Network, which contains two CNNs that connect to a NN. This model has been built, based on a model present in work [37] and gets better results than the other strategies, but without using cross-validation.

Thus, the following graphics show the results of scoring metrics and performance of encoding strategy. This procedure is performed for all scoring metrics, training and classification time, in order to perform a comparison of these results with the ones obtained in Figure 2.3 and in Figure 2.4.

The graphic on Figure 5.4 makes a comparison of the results obtained on CNN model and Complex Network model. There are some tables related to these results present in *Appendix* section. The graphic states that the best strategy is the 65 One-Hot vector. On almost all datasets is the best strategy too. The reason for this encoding strategy gets the best results may be to have an ideal amount of data and maybe the fact that all vectors are different from each other. The other strategies, such as One-Hot Amino Acid vector, Venn diagram and Amino Acid properties obtained a score close to the 65 One-Hot vector encode. Relatively to the strategy that uses the genetic sequence without codon format (One-Hot DNA sequence) the results obtained are a little low, because we have too much data pre-processed and then may be not well-trained.

The training and classification time results are obtained by the same way of score results, i.e., it is performed the mean of training and classification time on all datasets, and then in the graphic below is present the average of these results for each strategy.



	Physical chemical properties	One-Hot Amino Acid	Amino acid properties	Venn Diagram	65 One-Hot vector	One-Hot DNA seq
training	82,784	199,006	201,652	203,103	260,982	374,186
classification	1,721	4,819	4,105	4,225	8,001	9,686

Figure 5.3: Training and classification time for adopted encoding strategies (average for 10 folds). Pre-processing time is between 2 and 11 seconds

In the strategy One-Hot DNA sequence, the training time is bigger than the other strategies, because contains more amount of data than the others. Relating to Physical-chemical properties strategy, the fact that this strategy does not consist in vector encoding contributes to a low training time. Its data is composed by decimal values. All the remaining strategies have an encoding strategy based on vectors, which takes the time for training and classification bigger than the Physical-chemical properties strategy.

Another important note to take into account in these results is the size of the datasets. As we know, Humvar dataset has 40000 samples, and obviously takes much longer time in

training and classification. These results have been obtained by splitting the data in this way: 50% for training and the remaining 50% for testing, and then each split runs 10 times, because it is hard to determine the time of training and classification when using the scikit learn's cross-validation function *cross_val_score*.

In Complex Network results, the best score has been obtained for 65 One-Hot vector encode strategy. Results nearly 0.8 or more may be considered as good results, because the probability to perform well a prediction is high, i.e., the model that gets this result can be applied in pathogenicity prediction. In Figure A (in *Appendix* section) are present the results obtained from Complex Network on each dataset. The application of this Complex Network model using these strategies contributes a lot on getting good results. We can say that these model may be applied in the bioinformatics area to perform pathogenicity predictions.

In the Figure A, the best results are obtained on Varibench dataset. The drop of AUC happens mainly because of the dataset Swissvar and the strategy Physical-chemical properties. We can say that the strategy **Amino acid physical-chemical properties** is not a good strategy.

Then, are present a graphic and a table. The graphic presents the comparison between the AUC score obtained on CNN model and on Complex Network model. The table presents the comparison of best AUC scoring metric obtained on this dissertation with the obtained on the State of the Art (chapter 2).

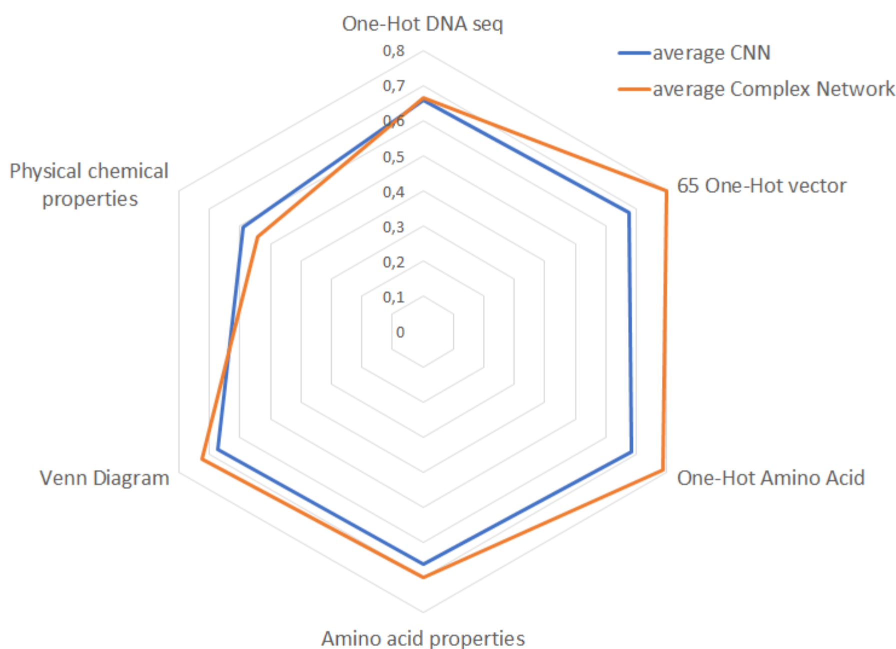


Figure 5.4: Accuracy score comparison between CNN and Complex Network model

		State of the art	
Dataset	*Best AUC*	*Best AUC*	*Non-biased*
Exovar	*0,83*	*0,92*	*0,83*
Humvar	*0,83*	*0,93*	*0,88*
PredictSNP	*0,78*	*0,93*	*0,80*
Swissvar	*0,74*	*0,73*	*0,73*
Varibench	*0,92*	*0,94*	*0,70*

Table 5.2: Comparison of AUC between CNN and Complex Network models with the State of the Art

Taking into account the radar graphic, we can say that the results are better when using the Complex Network model than the CNN. The reason behind this may be the number of hidden layers, neurons per layer, filters, the kernel size and batch size. It is also possible to see that 65 One-Hot vector and One-Hot Amino Acid vector strategies are those that get better results. Relating to the table, the best AUC obtained is on Varibench dataset, but these results are a little below comparing with those obtained on the State of the Art. Comparing with the "Non-biased" AUC results, they are better on two datasets.

Once analyzed these results, we can say that they were a little below of expectations, but interesting to be explored and applied in bioninformatics future work.

After this comparison, is present the training and classification time of the Complex Network model taking into account these results:

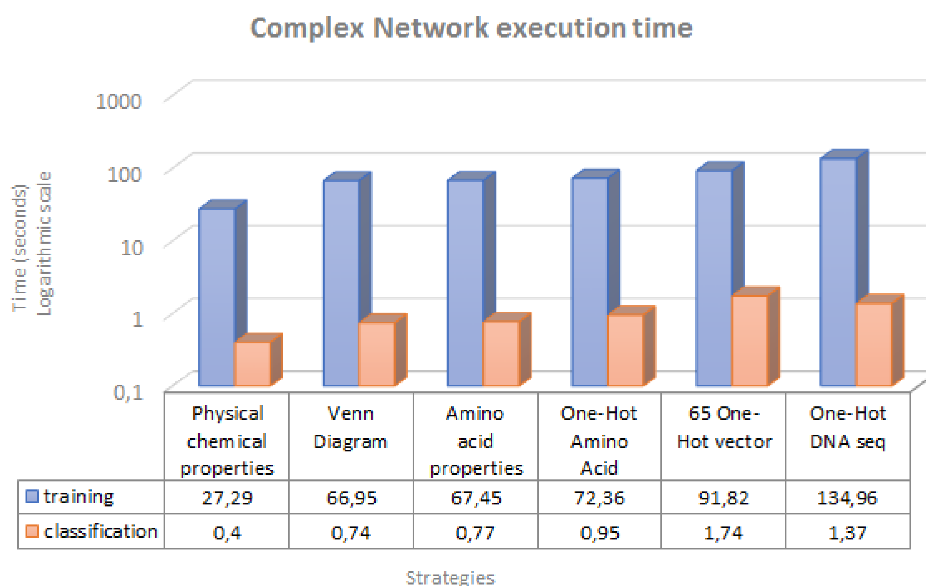


Figure 5.5: Training and classification time for Complex Network model (average for 10 folds). Pre-processing time is between 2 and 11 seconds

In this graphic are present the training and classification time results obtained by the

average of these times on each dataset. The strategy One-Hot DNA sequence is the strategy which takes more time to perform training and classification on all datasets, due to the amount of data. As stated in chapter 4, in this strategy each nucleotide present on the sequence is encoded by 5 One-Hot vector, i.e., each letter (nucleotide) is represented by One-Hot vector different among of them.

Obviously that these results are influenced by the size of the datasets. Humvar dataset takes a long time to train the model due to its amount of data. Amino acid physical-chemical properties strategy is the fastest one because of its encoding, i.e., does not use any vectors to encode the data. The remaining strategies takes more time on the datasets, because they use vectors to encode the data. The pre-processing time in Humvar takes nearly 11 seconds and in the remaining datasets takes between 2 and 4 seconds. Relating to the classification time, in Humvar dataset it takes nearly 5 seconds and in the remaining datasets between 1 and 2 seconds. There's an important fact to taking into account about classification time. Unlike the other training and classification time graphics, these results are obtained without cross-validation.

Chapter 6

Conclusions and Future Work

Technology area has been the area that obtained a great evolution in the past 20, 30 years. Nowadays, people can connect with each other easily, know where they are, even if they are far away from each other. This evolution has been verifying in several areas too, mainly in bioinformatics area. This area aims to improve people's quality of life collecting the respective biological data and then performing the analysis on genetic code, allowing people to know if a mutation or change detected in the human genetic code may cause a change in DNA sequence, or even cause an disease which may be transmitted for the next generations. This is an important situation for people's life because with a good prediction, the people will take care more about yourself in order to reduce the probability to contract a disease and avoid that the next generations from suffering the consequences.

So, this document presents an overview in machine learning and its sub area deep learning, creating models and use some classifiers in order to perform predictions on the human genetic code. These predictions are performed in order to know if a mutation in the genetic code may be pathogenic (cause a disease) or not. Based on recent studies performed in bio informatics area using some tools, this document starts by talking about these tools and in machine learning and deep learning contents. There are provided five datasets containing feature values about the genes, and then these datasets are used to perform the respective prediction, starting to use the classical classifiers in order to know if they perform well on the datasets. These tests are performed using k-fold cross-validation in order to avoid over-fitting, and only with this approach it is possible to know what are the classifiers that performs well. Also, are calculated the score for each scoring metric to help in evaluation of the classifier.

In the next phase, the format of datasets is different, that is, instead of having gene's feature values, we have the format presented in DNA sequence with the letters representing each nucleotide. They are encoded by using a strategy called One-Hot vector encode, in which

each letter is a 5 One-Hot vector. This approach is also used when the content of the datasets are in codons format. With this format, are applied different strategies to encode the data in order to perform the predictions and obtain results of scoring and performance (training and classification time).

Lastly is performed an analysis of the results obtained for scoring metrics and for the performance of each classifier and models that use some strategies. These results and analysis are present in chapter 5. Once this analysis is done, we can compare the results presented on this document with the results obtained in some works stated in chapter 2. Has been used five datasets on this dissertation (Exovar, Humvar, Swissvar, PredictSNP and Varibench). Comparing the results obtained on this dissertation with the Figure 2.3 and Figure 2.4 (chapter 2), we can say that the Complex Network model is better than the tools used in Figure 2.4 (CADD, GWAVA, FatHMM, etc) and is also better than some other tools used on Figure 2.3. Relating to the results obtained on the CNN model with cross-validation, they are similar to those on Figure 2.4 and none of them are better than the tools used on Figure 2.3. This means that some of them used in chapter 2 are better than the models created in this dissertation, but these models are better than a lot of tools used in some works performed in bioinformatics area, mainly the Complex Network model.

Relating to the score and performance results obtained, we can say that the score may depend from the size of the dataset, type of the data and on the values of the arguments used by each model. For training and classification time, the results may depend on the number of samples of a dataset and its type of content, i.e, in this case the encoding strategy.

Although the results stayed a little below expectations, these models can be improved in the near future to perform predictions and originate new prediction models.

This analysis can prove that the models created and the strategies adopted are interesting to apply them in bioinformatics area and in a web tool, where could perform an upload of a script containing the selected dataset and then the script with the models and respective strategies run on a simulator to perform the predictions. Due to time constraints, the creation of this web platform could not be implemented. This also happened because over the work, we focused more on get better results of pathogenicity prediction. To aim this, the focus turned to the creation of several models and strategies adopted over the models, and on the training and classification time obtained on these models.

Finally, as already stated, an incorporation of these models and strategies approached in this dissertation in a web tool similar to the one mentioned in chapter 2, using Representational State Transfer (REST) services, is a good target to future work.

Bibliography

[1] R. Eklund and A. Rautanen, "582314 G e n e t i c s f o r C o m p u t e r S c i e n t i s t s , 2 CU Course assistants : Organizers :", pp. 1–56, 2004. [Online]. Available: https://www.cs.helsinki.fi/group/genetics/Genetics_for_CS_March_04.pdf.

[2] D. W. Fawcett, *Nucleus*. 1966, pp. 195–302, ISBN: 0721635849. [Online]. Available: https://www.ascb.org/wp-content/uploads/2015/12/FawcettTheCellChapter4.pdf.

[3] D. G. Grimm, C. A. Azencott, F. Aicheler, U. Gieraths, D. G. Macarthur, K. E. Samocha, D. N. Cooper, P. D. Stenson, M. J. Daly, J. W. Smoller, L. E. Duncan, and K. M. Borgwardt, "The evaluation of tools used to predict the impact of missense variants is hindered by two types of circularity", *Human Mutation*, vol. 36, no. 5, pp. 513–523, 2015, ISSN: 10981004. DOI: 10.1002/humu.22768.

[4] H. Tang and P. D. Thomas, "PANTHER-PSEP: Predicting disease-causing genetic variants using position-specific evolutionary preservation", *Bioinformatics*, vol. 32, no. 14, pp. 2230–2232, 2016, ISSN: 14602059. DOI: 10.1093/bioinformatics/btw222.

[5] J. Bendl, M. Musil, J. Štourač, J. Zendulka, J. Damborský, and J. Brezovský, "PredictSNP2: A Unified Platform for Accurately Evaluating SNP Effects by Exploiting the Different Characteristics of Variants in Distinct Genomic Regions", *PLoS Computational Biology*, vol. 12, no. 5, 2016, ISSN: 15537358. DOI: 10.1371/journal.pcbi.1004962.

[6] I. A. Adzhubei, S. Schmidt, L. Peshkin, V. E. Ramensky, A. Gerasimova, P. Bork, A. S. Kondrashov, and S. R. Sunyaev, "A method and server for predicting damaging missense mutations", *Nature Methods*, vol. 7, no. 4, pp. 248–249, 2010, ISSN: 1548-7091. DOI: 10.1038/nmeth0410-248. arXiv: 0601019 [q-bio]. [Online]. Available: http://dx.doi.org/10.1038/nmeth0410-248.

[7] P. C. Ng and S. Henikoff, "SIFT: Predicting amino acid changes that affect protein function", *Nucleic Acids Research*, vol. 31, no. 13, pp. 3812–3814, 2003, ISSN: 03051048. DOI: 10.1093/nar/gkg509.

[8] E. Lubeck, A. F. Coskun, T. Zhiyentayev, M. Ahmad, and L. Cai, "MutationTaster2: mutation prediction for the deep-sequencing age", *Nat. Methods Nat. Methods Nat. Methods*, vol. 9, no. 10, pp. 743–748, 2012, ISSN: 1548-7091. DOI: 10.1038/nmeth.2892. [Online]. Available: http://dx.doi.org/10.1038/nmeth.2890.

[9] B. Reva, Y. Antipin, and C. Sander, "Predicting the functional impact of protein mutations: Application to cancer genomics", *Nucleic Acids Research*, vol. 39, no. 17, 2011, ISSN: 03051048. DOI: 10.1093/nar/gkr407.

[10] M. Kircher, D. M. Witten, P. Jain, B. J. O'Roak, G. M. Cooper, J. Shendure, B. J. O. Roak, G. M. Cooper, and J. Shendure, "A general framework for estimating the relative pathogenicity of human genetic variants.", *Nature Genetics*, vol. 46, no. 3, pp. 310–315, 2014, ISSN: 1546-1718. DOI: 10.1038/ng.2892. arXiv: NIHMS150003. [Online]. Available: http://dx.doi.org/10.1038/ng.2892%7B%5C%%7D5Cnhttp://www.ncbi.nlm.nih.gov/pubmed/24487276.

[11] S. Chun and J. C. Fay, "Identification of deleterious mutations within three human genomes", *Genome Research*, vol. 19, no. 9, pp. 1553–1561, 2009, ISSN: 10889051. DOI: 10.1101/gr.092619.109.

[12] E. V. Davydov, D. L. Goode, M. Sirota, G. M. Cooper, A. Sidow, and S. Batzoglou, "Identifying a high fraction of the human genome to be under selective constraint using GERP++", *PLoS Computational Biology*, vol. 6, no. 12, 2010, ISSN: 1553734X. DOI: 10.1371/journal.pcbi.1001025.

[13] H. A. Shihab, J. Gough, D. N. Cooper, P. D. Stenson, G. L. A. Barker, K. J. Edwards, I. N. M. Day, and T. R. Gaunt, "Predicting the Functional, Molecular, and Phenotypic Consequences of Amino Acid Substitutions using Hidden Markov Models", *Human Mutation*, vol. 34, no. 1, pp. 57–65, 2013, ISSN: 10597794. DOI: 10.1002/humu.22225.

[14] G. M. Cooper and J. Shendure, "Needles in stacks of needles: finding disease-causal variants in a wealth of genomic data.", *Nature reviews. Genetics*, vol. 12, no. 9, pp. 628–40, 2011, ISSN: 1471-0064. DOI: 10.1038/nrg3046. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/21850043.

[15] M. X. Li, J. S. H. Kwan, S. Y. Bao, W. Yang, S. L. Ho, Y. Q. Song, and P. C. Sham, "Predicting Mendelian Disease-Causing Non-Synonymous Single Nucleotide Variants in Exome Sequencing Studies", *PLoS Genetics*, vol. 9, no. 1, 2013, ISSN: 15537390. DOI: 10.1371/journal.pgen.1003143.

[16] Y. Fu, Z. Liu, S. Lou, J. Bedford, X. J. Mu, K. Y. Yip, E. Khurana, M. Gerstein, T. P. Pdf, G. Biology, Y. Fu, Z. Liu, S. Lou, J. Bedford, X. J. Mu, E. Khurana, M. Gerstein, I. Article, A. Url, P. Central, and B. Central, "FunSeq2: A framework for prioritizing noncoding regulatory variants in cancer", *Genome Biology*, vol. 15, no. 10, p. 480, 2014, ISSN: 1465-6906. DOI: 10.1186/s13059-014-0480-5. [Online]. Available: http://genomebiology.com/2014/15/10/480.

[17] E. A. Stone and A. Sidow, "Physicochemical constraint violation by missense substitutions mediates impairment of protein function and disease severity", *Genome Research*, vol. 15, no. 7, pp. 978–986, 2005, ISSN: 10889051. DOI: 10.1101/gr.3804205.

[18] E. Capriotti, R. Calabrese, and R. Casadio, "Predicting the insurgence of human genetic diseases associated to single point protein mutations with support vector machines and evolutionary information", *Bioinformatics*, vol. 22, no. 22, pp. 2729–2734, 2006, ISSN: 13674803. DOI: 10.1093/bioinformatics/btl423.

[19] V. Ramensky, P. Bork, and S. Sunyaev, "Human non-synonymous SNPs: server and survey.", *Nucleic acids research*, vol. 30, no. 17, pp. 3894–3900, 2002, ISSN: 1362-4962. DOI: 10.1093/nar/gkf493.

[20] Y. Bromberg and B. Rost, "SNAP: Predict effect of non-synonymous polymorphisms on function", *Nucleic Acids Research*, vol. 35, no. 11, pp. 3823–3835, 2007, ISSN: 03051048. DOI: 10.1093/nar/gkm238. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1920242/.

[21] J. Bendl, J. Stourac, O. Salanda, A. Pavelka, E. D. Wieben, J. Zendulka, J. Brezovsky, and J. Damborsky, "PredictSNP: Robust and Accurate Consensus Classifier for Prediction of Disease-Related Mutations", *PLoS Computational Biology*, vol. 10, no. 1, 2014, ISSN: 1553734X. DOI: 10.1371/journal.pcbi.1003440.

[22] K. Wang, M. Li, and H. Hakonarson, "ANNOVAR: functional annotation of genetic variants from high-throughput sequencing data", *Nucleic acids research*, vol. 38, no. 16, e164, 2010, ISSN: 13624962. DOI: 10.1093/nar/gkq603.

[23] H. a. Shihab, M. F. Rogers, J. Gough, M. Mort, D. N. Cooper, I. N. M. Day, T. R. Gaunt, and C. Campbell, "An integrative approach to predicting the functional effects of non-coding and coding sequence variation.", *Bioinformatics (Oxford, England)*, vol. 31, no. January, pp. 1536–1543, 2015, ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btv009. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/25583119.

[24] G. R. S. Ritchie, I. Dunham, E. Zeggini, and P. Flicek, "Functional annotation of noncoding sequence variants.", *Nature methods*, vol. 11, no. 3, pp. 294–6, 2014, ISSN: 1548-7105. DOI: 10.1038/nmeth.2832. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/24487584.

[25] M. J. Landrum, J. M. Lee, G. R. Riley, W. Jang, W. S. Rubinstein, D. M. Church, and D. R. Maglott, "ClinVar: Public archive of relationships among sequence variation and human phenotype", *Nucleic Acids Research*, vol. 42, no. D1, 2014, ISSN: 03051048. DOI: 10.1093/nar/gkt1113.

[26] S. T. Sherry, M. H. Ward, M. Kholodov, J. Baker, L. Phan, E. M. Smigielski, and K. Sirotkin, "dbSNP: the NCBI database of genetic variation.", *Nucleic acids research*, vol. 29, no. 1, pp. 308–311, 2001, ISSN: 1362-4962. DOI: 10.1093/nar/29.1.308. arXiv: /www.pubmedcentral.nih.gov/articlerender.fc. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/11125122.

[27] F. Cunningham, M. R. Amode, D. Barrell, K. Beal, K. Billis, S. Brent, D. Carvalho-Silva, P. Clapham, G. Coates, S. Fitzgerald, L. Gil, C. G. Girón, L. Gordon, T. Hourlier, S. E. Hunt, S. H. Janacek, N. Johnson, T. Juettemann, A. K. Kähäri, S. Keenan, F. J. Martin, T. Maurel, W. McLaren, D. N. Murphy, R. Nag, B. Overduin, A. Parker, M. Patricio, E. Perry, M. Pignatelli, H. S. Riat, D. Sheppard, K. Taylor, A. Thormann, A. Vullo, S. P. Wilder, A. Zadissa, B. L. Aken, E. Birney, J. Harrow, R. Kinsella, M. Muffato, M. Ruffier, S. M. J. Searle, G. Spudich, S. J. Trevanion, A. Yates, D. R. Zerbino, and P. Flicek, "Ensembl 2015", *Nucleic Acids Research*, vol. 43, no. D1, pp. D662–D669, 2015, ISSN: 13624962. DOI: 10.1093/nar/gku1010.

[28] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers, "GenBank", *Nucleic Acids Research*, vol. 39, no. SUPPL. 1, 2011, ISSN: 03051048. DOI: 10.1093/nar/gkq1079.

[29] L. D. Ward and M. Kellis, "HaploReg v4: Systematic mining of putative causal variants, cell types, regulators and target genes for human complex traits and disease", *Nucleic Acids Research*, vol. 44, no. D1, pp. D877–D881, 2016, ISSN: 13624962. DOI: 10.1093/nar/gkv1340.

[30] J. S. Amberger, C. A. Bocchini, F. Schiettecatte, A. F. Scott, and A. Hamosh, "OMIM.org: Online Mendelian Inheritance in Man (OMIM??), an Online catalog of human genes and genetic disorders", *Nucleic Acids Research*, vol. 43, no. D1, pp. D789–D798, 2015, ISSN: 13624962. DOI: 10.1093/nar/gku1205.

[31] A. P. Boyle, E. L. Hong, M. Hariharan, Y. Cheng, M. A. Schaub, M. Kasowski, K. J. Karczewski, J. Park, B. C. Hitz, S. Weng, J. M. Cherry, and M. Snyder, "Annotation of functional variation in personal genomes using RegulomeDB", *Genome Research*, vol. 22, no. 9, pp. 1790–1797, 2012, ISSN: 10889051. DOI: 10.1101/gr.137323.112.

[32] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and a. D. Haussler, "The Human Genome Browser at UCSC", *Genome Research*, vol. 12, no. 6, pp. 996–1006, 2002, ISSN: 1088-9051. DOI: 10.1101/gr.229102. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/12045153.

[33] Paulo Miguel da Silva Gaspar, "Machine Learning and Probabilistic Models for Biomedical Knowledge Engineering", 2014. [Online]. Available: http://ria.ua.pt/handle/10773/13949.

[34] Z. Wang and C. B. Burge, "Splicing regulation: from a parts list of regulatory elements to an integrated splicing code.", *RNA (New York, N.Y.)*, vol. 14, no. 5, pp. 802–13, 2008, ISSN: 1469-9001. DOI: 10.1261/rna.876308. arXiv: NIHMS150003. [Online]. Available: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2327353%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract%7B%5C%%7D5Cnhttp://www.ncbi.nlm.nih.gov/pubmed/18369186%7B%5C%%7D5Cnhttp://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC2327353.

[35] Liam J. McGuffin, K. Bryson, and D. T. Jones, "The PSIPRED protein structure prediction server", *Bioinformatics (Oxford, England)*, vol. 16, no. 4, pp. 404–405, 2000, ISSN: 1367-4803. DOI: 10.1093/bioinformatics/16.4.404. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/10869041.

[36] A. Karpathy, L. Fei-Fei, and J. Johnson, *CS231n Convolutional Neural Networks for Visual Recognition*, 2016. [Online]. Available: http://cs231n.github.io/convolutional-networks/.

[37] C. Angermueller, T. Pärnamaa, L. Parts, and S. Oliver, "Deep Learning for Computational Biology", *Molecular Systems Biology*, no. 12, p. 878, 2016, ISSN: 1744-4292. DOI: 10.15252/msb.20156651.

[38] C. Murray, "Deep Learning CNN's in Tensorflow with GPUs", [Online]. Available: https://hackernoon.com/deep-learning-cnns-in-tensorflow-with-gpus-cba6efe0acc2.

[39] B. Alipanahi, A. Delong, M. T. Weirauch, and B. J. Frey, "Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning.", *Nat Biotechnol*, vol. 33, no. 8, pp. 831–838, 2015, ISSN: 1087-0156. DOI: 10.1038/nbt.3300. arXiv: 9605103 [cs]. [Online]. Available: http://dx.doi.org/10.1038/nbt.3300.

[40] D. R. Kelley, J. Snoek, and J. L. Rinn, "Basset: Learning the regulatory code of the accessible genome with deep convolutional neural networks", *Genome Research*, vol. 26, no. 7, pp. 990–999, 2016, ISSN: 15495469. DOI: 10.1101/gr.200535.115.

[41] F. Ning, D. Delhomme, Y. LeCun, F. Piano, L. Bottou, and P. E. Barbano, "Toward automatic phenotyping of developing embryos from videos", *IEEE Transactions on Image Processing*, vol. 14, no. 9, pp. 1360–1371, 2005, ISSN: 10577149. DOI: 10.1109/TIP.2005.852470.

[42] D. C. Cirefffdfffdan, A. Giusti, L. M. Gambardella, and J. Schmidhuber, "Mitosis detection in breast cancer histology images with deep neural networks", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8150 LNCS, 2013, pp. 411–418, ISBN: 9783642407628. DOI: 10.1007/978-3-642-40763-5_51.

[43] V. Kovalev, A. Kalinovsky, and S. Kovalev, "Deep Learning with Theano , Torch , Caffe , TensorFlow , and Deeplearning4J : Which One Is the Best in Speed and Accuracy ?", no. October, pp. 99–103, 2016. [Online]. Available: https://www.researchgate.net/publication/302955130_Deep_Learning_with_Theano_Torch_Caffe_TensorFlow_and_Deeplearning4J_Which_One_Is_the_Best_in_Speed_and_Accuracy.

[44] F. Dernoncourt, *Classification - What does AUC stand for and what is it? - Cross Validated*, 2015. [Online]. Available: https://stats.stackexchange.com/questions/132777/what-does-auc-stand-for-and-what-is-it (visited on 06/06/2017).

[45] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", in *Proceedings of the IEEE International Conference on Computer Vision*, vol. 11-18-Dece, Feb. 2016, pp. 1026–1034, ISBN: 9781467383912. DOI: 10.1109/ICCV.2015.123. arXiv: 1502.01852. [Online]. Available: http://arxiv.org/abs/1502.01852.

[46] N. Giang Nguyen, V. A. Tran, D. L. Ngo, D. Phan, F. R. Lumbanraja, M. R. Faisal, B. Abapihi, M. Kubo, and K. Satou, "DNA Sequence Classification by Convolutional Neural Network", *J. Biomedical Science and Engineering*, vol. 9, no. 9, pp. 280–286, 2016, ISSN: 1937-6871. DOI: 10.4236/jbise.2016.95021. [Online]. Available: http://www.scirp.org/JOURNAL/PaperInformation.aspx?PaperID=65923.

[47] J. Kyte and R. F. Doolittle, "A simple method for displaying the hydropathic character of a protein", *Journal of Molecular Biology*, vol. 157, no. 1, pp. 105–132, 1982, ISSN: 00222836. DOI: 10.1016/0022-2836(82)90515-0. [Online]. Available: http://www.biosyn.com/Images/ArticleImages/pdf/A%20simple.pdf.

[48] E. A. Zanaty, "Support Vector Machines (SVMs) versus Multilayer Perception (MLP) in data classification", *Egyptian Informatics Journal*, vol. 13, no. 3, pp. 177–183, 2012, ISSN: 11108665. DOI: 10.1016/j.eij.2012.08.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1110866512000345.

Appendix

Classical classifiers results

	ExoVar			HumVar		
Classifiers	*Accuracy*	*AUC*	*MCC*	*Accuracy*	*AUC*	*MCC*
AdaBoost	*0.7942*	*0.8628*	*0.5782*	*0.7847*	*0.8615*	*0.5688*
Decision Tree	*0.6770*	*0.6886*	*0.3736*	*0.6511*	*0.6577*	*0.3282*
KNeighbors	*0.7683*	*0.8337*	*0.5216*	*0.7607*	*0.8410*	*0.5246*
Log Regression	*0.8032*	*0.8708*	*0.5952*	*0.7888*	*0.8645*	*0.5768*
Naive Bayes	*0.7126*	*0.8086*	*0.4253*	*0.6284*	*0.7955*	*0.3168*
Neural Network	*0.7763*	*0.8553*	*0.4253*	*0.7325*	*0.8124*	*0.4818*
Random Forest	*0.7331*	*0.8232*	*0.4860*	*0.6844*	*0.8118*	*0.4161*
SVM	*0.8065*	***0.8731***	*0.6002*	*0.8132*	***0.8982***	*0.6198*

Table A.1: Exovar and Humvar results

	PredictSNP			Swissvar		
Classifiers	*Accuracy*	*AUC*	*MCC*	*Accuracy*	*AUC*	*MCC*
AdaBoost	*0.7954*	*0.8566*	*0.5622*	*0.6717*	*0.6667*	*0.2149*
Decision Tree	*0.6844*	*0.6966*	*0.3814*	*0.5885*	*0.5471*	*0.0943*
KNeighbors	*0.7748*	*0.8329*	*0.5191*	*0.6491*	*0.6345*	*0.1362*
Log Regression	*0.8096*	*0.8744*	*0.5901*	*0.6716*	*0.6874*	*0.2279*
Naive Bayes	*0.6293*	*0.6078*	*0.068*	*0.3688*	*0.5202*	*0.0338*
Neural Network	*0.7483*	*0.7483*	*0.4957*	*0.6347*	*0.636*	*0.1818*
Random Forest	*0.7176*	*0.8219*	*0.46*	*0.6451*	*0.625*	*0.1413*
SVM	*0.8242*	***0.8963***	*0.6234*	*0.7429*	***0.7532***	*0.4451*

Table A.2: PredictSNP and Swissvar results

	Varibench		
Strategies	*Accuracy*	*AUC*	*MCC*
AdaBoost	*0.8289*	*0.8244*	*0.4796*
DecisionTree	*0.776*	*0.7617*	*0.3294*
KNeighbors	*0.8381*	*0.8921*	*0.6225*
Log Regression	*0.8481*	*0.8446*	*0.5371*
Naive Bayes	*0.4781*	*0.5723*	*0.1661*
Neural Network	*0.8395*	*0.8295*	*0.5092*
Random Forest	*0.8013*	*0.7798*	*0,4077*
SVM	*0.8315*	***0.9021***	*0.6632*

Table A.3: Varibench results

	Exovar		Humvar	
Classifiers	*training*	*classification*	*training*	*classification*
AdaBoost	*4.42s*	*0.13s*	*22.67s*	*0.63s*
DecisionTree	*2.91s*	*0.01s*	*14.94s*	*0.07s*
KNeighbors	*0.17s*	*30.29s*	*1.41s*	*5m 5s*
Logistic Regression	*10.54s*	*0.008s*	*2m 11s*	*0.04s*
Naive Bayes	*0.08s*	*0.07s*	*0.39s*	*0.36s*
Neural Network	*18.41s*	*0,47s*	*1m 4s*	*2,02s*
Random Forest	*17.50s*	*0.07s*	*96.23s*	*0.39s*
SVM	*1m 53s*	*1.83s*	*4m 20s*	*3.45s*

Table A.4: Exovar and Humvar execution time

	PredictSNP		**Swissvar**	
Classifiers	*training*	*classification*	*training*	*classification*
AdaBoost	*7.72s*	*0.22s*	*6.68s*	*0.21s*
DecisionTree	*4.71s*	*0.02s*	*3.78s*	*0.03s*
KNeighbors	*0.36s*	*49.43s*	*0.26s*	*36.73s*
Logistic Regression	*24.09s*	*0.02s*	*19.71s*	*0.014s*
Naive Bayes	*0.17s*	*0.15s*	*0.11s*	*0.12s*
Neural Network	*26.50s*	*0.76s*	*22.46s*	*0.67s*
Random Forest	*30.83s*	*0.33s*	*23.58s*	*0.11s*
SVM	*2m 25s*	*2.35s*	*2m 10s*	*2.12s*

Table A.5: PredictSNP and Swissvar execution time

	Varibench	
Classifiers	*training*	*classification*
AdaBoost	*5.07s*	*0.16s*
DecisionTree	*2.84s*	*0.01s*
KNeighbors	*0.23s*	*22.91s*
Logistic Regression	*14.42s*	*0.01s*
Naive Bayes	*0.09s*	*1,09s*
Neural Network	*20.98s*	*0.52s*
Random Forest	*16.46s*	*0.07s*
SVM	*2m*	*1.96s*

Table A.6: Varibench execution time

CNN with AUC and execution time results

Strategies	**Exovar**	**Humvar**	**PredictSNP**	**Swissvar**	**Varibench**
One-Hot DNA sequence	*0.5837*	*0.6932*	*0.7283*	*0.6145*	*0.6742*
65 One-Hot vector	*0.6856*	*0.7118*	*0.7346*	*0.6017*	*0.642*
One-Hot amino acid	*0.6955*	*0.7127*	*0.7379*	*0.6013*	*0.6732*
Amino acid properties	*0.6693*	*0.685*	*0.7175*	*0.5869*	*0.6523*
Venn diagram	*0.6898*	*0.7071*	*0.735*	*0.5988*	*0.6374*
Physical-chemical properties	*0.6862*	*0.7069*	*0.5566*	*0.5005*	*0.5066*

Table A.7: AUC score of Convolutional Neural Network

	Exovar		Humvar	
Strategies	*training*	*classification*	*training*	*classification*
One-Hot DNA seq	*3m 58s*	*5.73s*	*14m 4s*	*23.33s*
65 One-Hot vec	*2m 2s*	*4.67s*	*9m 28s*	*17.64s*
One-Hot amino acid	*1m 52s*	*2.49s*	*7m 18s*	*11.37s*
Amino acid prop	*1m 45s*	*1.99s*	*7m 49s*	*9.74s*
Venn diagram	*1m 45s*	*2.76s*	*7m 51s*	*9.19s*
Physical-chemical	*40.88s*	*0.88s*	*3m 4s*	*3.82s*

Table A.8: Exovar and Humvar DNA sequence execution time

	PredictSNP		Swissvar	
Strategies	*training*	*classification*	*training*	*classification*
One-Hot DNA seq	*5m 4s*	*8.36s*	*4m 21s*	*6.17s*
65 One-Hot vec	*3m 40s*	*6.70s*	*3m 44s*	*5.61s*
One-Hot amino acid	*2m 45s*	*3.97s*	*2m 21s*	*3.39s*
Amino acid prop	*2m 58s*	*3.39s*	*2m 25s*	*3.05s*
Venn diagram	*2m 52s*	*3.78s*	*2m 24s*	*3.04s*
Physical-chemical	*1m 9s*	*1.48s*	*59.68s*	*1.37s*

Table A.9: PredictSNP and Swissvar DNA sequence execution time

	Varibench	
Strategies	*training*	*classification*
One-Hot DNA seq	*3m 42s*	*4.84s*
65 One-Hot vec	*2m 50s*	*5.38s*
One-Hot amino acid	*2m 18s*	*2.88s*
Amino acid prop	*1m 51s*	*2.36s*
Venn diagram	*2m 3s*	*2.41s*
Physical-chemical	*59.98s*	*1.07s*

Table A.10: Varibench DNA sequence execution time

Complex Network with AUC and execution time results

Strategies	Exovar	Humvar	PredictSNP	Swissvar	Varibench
One-Hot DNA sequence	*0.6856*	*0.7397*	*0.6542*	*0.5424*	*0.695*
65 One-Hot vector	*0.832*	*0.8267*	*0.7818*	*0.6431*	*0.9168*
One-Hot amino acid	*0.804*	*0.7686*	*0.7558*	*0.7375*	*0.8814*
Amino acid properties	*0.7012*	*0.6929*	*0.6965*	*0.6173*	*0.797*
Venn diagram	*0.748*	*0.7449*	*0.7125*	*0.5871*	*0.835*
Physical-chemical properties	*0.5815*	*0.5032*	*0.5032*	*0.5312*	*0.5811*

Table A.11: AUC score of Complex Network

	Exovar		Humvar	
Strategies	*training*	*classification*	*training*	*classification*
One-Hot DNA sequence	*1m 10s*	*0.83s*	*5m 12s*	*3.62s*
65 One-Hot vector	*46.55s*	*0.79s*	*3m 33s*	*4.55s*
One-Hot amino acid	*37.01s*	*0.49s*	*2m 50s*	*2.13s*
Amino acid properties	*34.36s*	*0.41s*	*2m 40s*	*1.75s*
Venn diagram	*32.09s*	*0.01s*	*2m 41s*	*1.89s*
Physical-chemical properties	*14.81s*	*0.19s*	*1m 4s*	*0.68s*

Table A.12: Exovar and Humvar execution time results for Complex Network

	PredictSNP		Swissvar	
Strategies	*training*	*classification*	*training*	*classification*
One-Hot DNA sequence	*1m 56s*	*1.39s*	*1m 37s*	*0.04s*
65 One-Hot vector	*1m 21s*	*1.32s*	*1m 9s*	*1.12s*
One-Hot amino acid	*1m 3s*	*0.86s*	*51.62s*	*0.69s*
Amino acid properties	*56.22s*	*0.68s*	*48.97s*	*0.56s*
Venn diagram	*57.41s*	*0.69s*	*48.79s*	*0.62s*
Physical-chemical properties	*26.88s*	*0.62s*	*16.78s*	*0.29s*

Table A.13: PredictSNP and Swissvar execution time results for Complex Network

	Varibench	
Strategies	*training*	*classification*
One-Hot DNA sequence	*1m 19s*	*0.96s*
65 One-Hot vector	*48.58s*	*0.92s*
One-Hot amino acid	*39.90s*	*0.59s*
Amino acid properties	*37.43s*	*0.47s*
Venn diagram	*35.39s*	*0.50s*
Physical-chemical properties	*13.65s*	*0.22s*

Table A.14: Varibench execution time results for Complex Network

AUC score comparison between CNN and Complex Network

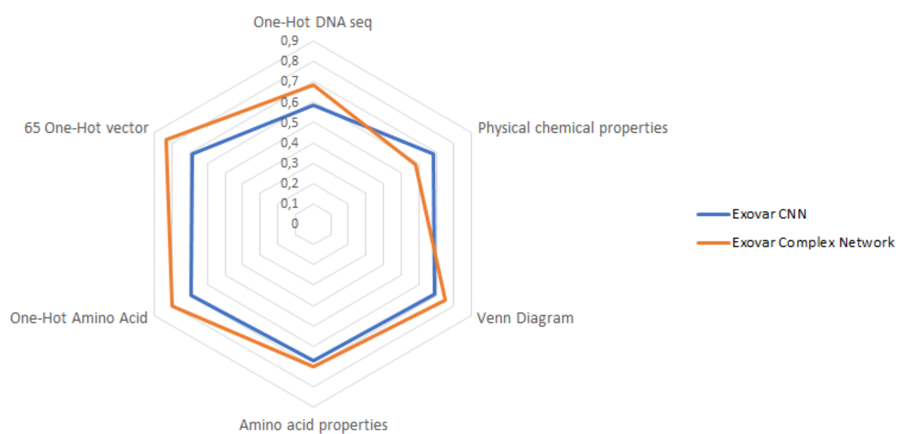


Figure A.1: Results for CNN and Complex Network model on Exovar dataset

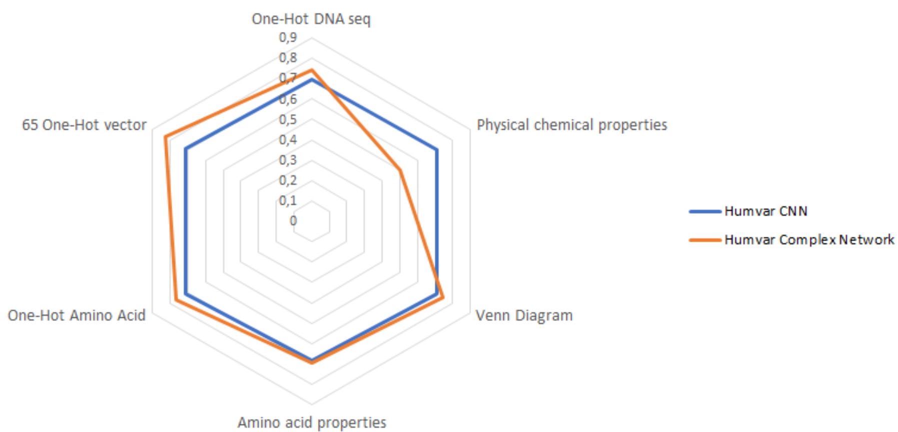


Figure A.2: Results for CNN and Complex Network model on Humvar dataset

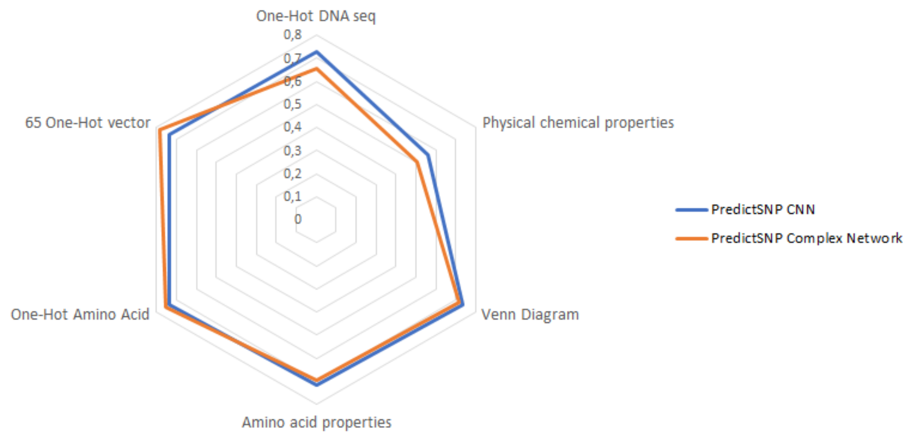


Figure A.3: Results for CNN and Complex Network model on PredictSNP dataset

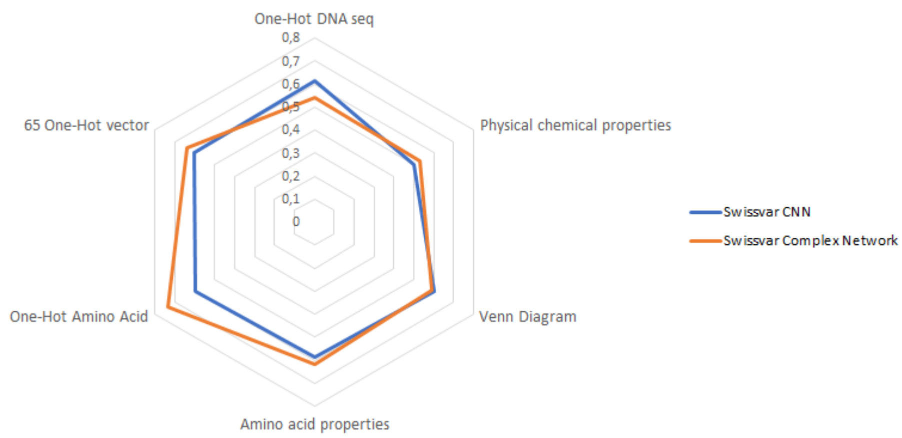


Figure A.4: Results for CNN and Complex Network model on Swissvar dataset

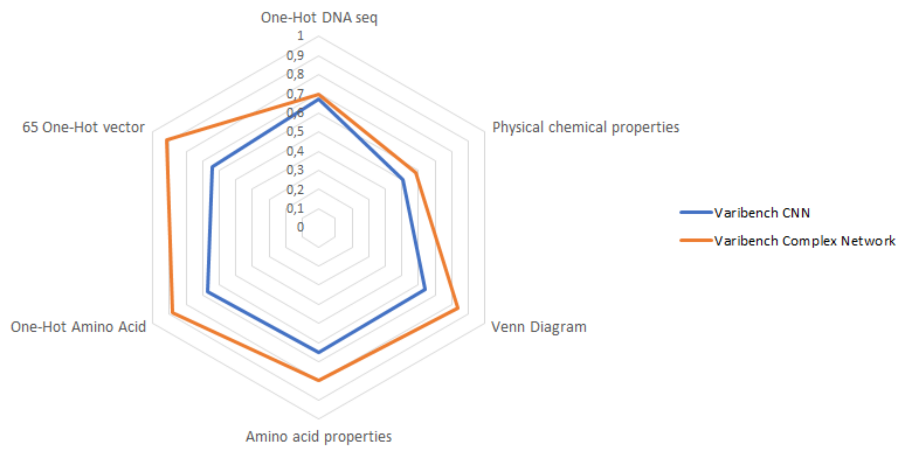


Figure A.5: Results for CNN and Complex Network model on Varibench dataset