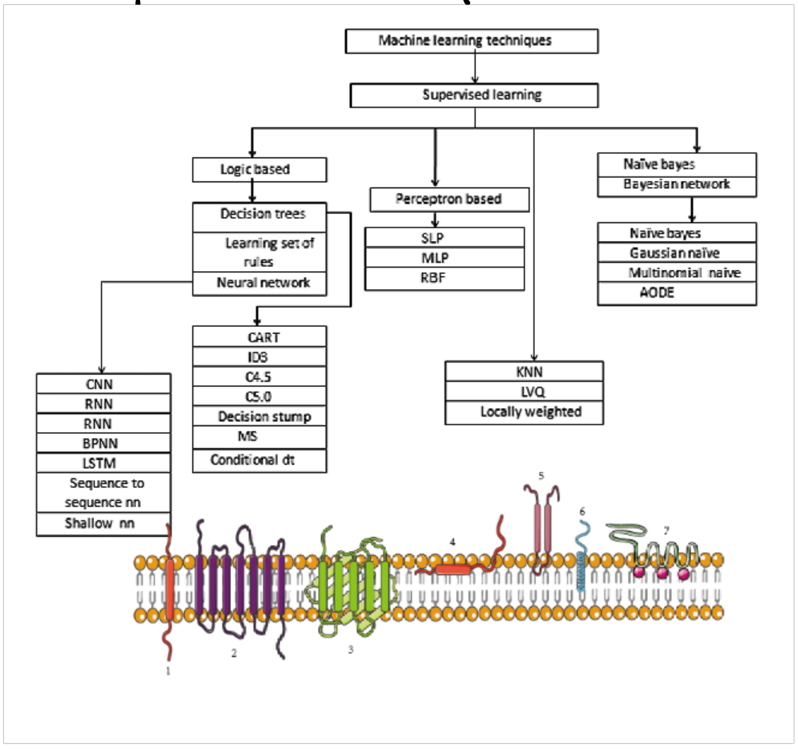
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικό και Καποδιστριακό
Πανεπιστήμιο Αθηνών

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΒΙΟΛΟΓΙΑΣ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
«ΒΙΟΠΛΗΡΟΦΟΡΙΚΗ - ΥΠΟΛΟΓΙΣΤΙΚΗ ΒΙΟΛΟΓΙΑ»

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

«Τεχνικές εξόρυξης γνώσης και εφαρμογές τους σε βιολογικές βάσεις δεδομένων»



Πέτρος-Ευάγγελος Ταμβάκης
Πτυχιούχος Φυσικών Επιστημών, Ε.Α.Π

ΑΘΗΝΑ, 2020

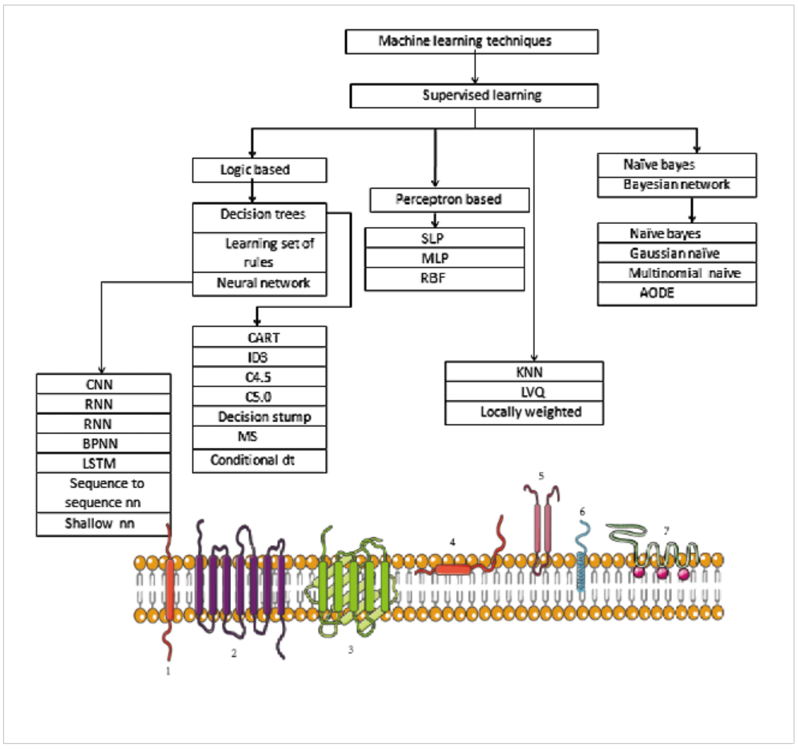
HELLENIC REPUBLIC
National and Kapodistrian
University of Athens

SCHOOL OF SCIENCE
FACULTY OF BIOLOGY

MASTER IN “BIOINFORMATICS”

MASTER DIPLOMA THESIS

«Data mining techniques and their applications in biological databases»



Petros - Evaggelos Tamvakis
BSc Natural Sciences, H.O.U

ATHENS, 2020

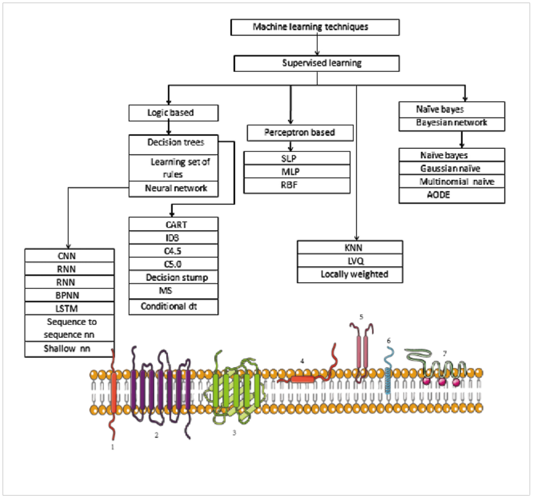
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικό και Καποδιστριακό
Πανεπιστήμιο Αθηνών

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΒΙΟΛΟΓΙΑΣ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ «ΒΙΟΠΛΗΡΟΦΟΡΙΚΗ - ΥΠΟΛΟΓΙΣΤΙΚΗ ΒΙΟΛΟΓΙΑ»

Δ Ι Π Λ Ω Μ Α Τ Ι Κ Η   Ε Ρ Γ Α Σ Ι Α

«Τεχνικές εξόρυξης γνώσης και εφαρμογές τους σε βιολογικές βάσεις δεδομένων»



Τριμελής Εξεταστική Επιτροπή

Αναπληρωτής Καθηγητής Μιχαήλ Φιλιππάκης (Επιβλέπων)

Σχολή Τεχνολογιών Πληροφορικής και Επικοινωνιών
Τμήμα Ψηφιακών Συστημάτων
Πανεπιστήμιο Πειραιώς

Καθηγητής Ιωάννης Τρουγκάκος

Τομέας Βιολογίας Κυττάρου και Βιοφυσικής
Τμήμα Βιολογίας, Εθνικό και Καποδιστριακό Πανεπιστήμιο

Αναπληρώτρια Καθηγήτρια Βασιλική Οικονομίδου

Τομέας Βιολογίας Κυττάρου και Βιοφυσικής
Τμήμα Βιολογίας, Εθνικό και Καποδιστριακό Πανεπιστήμιο

Ευχαριστώ ιδιαιτέρως τον επιβλέποντα καθηγητή κο Μιχαήλ Φιλιππάκη για την αδιάκοπτη βοήθεια, συνεργασία και συνεισφορά του στην ολοκλήρωση αυτής της διπλωματικής εργασίας.

Contents

1 Περίληψη

Σκοπός αυτής της διπλωματικής εργασίας είναι η κατασκευή και ανάπτυξη αλγορίθμων μηχανικής και βαθιάς μάθησης (που αποτελούν τον βασικό πυρήνα των τεχνικών εξόρυξης γνώσης) για την πρόγνωση και ταξινόμηση μεμβρανικών πρωτεΐνών και να αναφέρει την απόδοση τους. Αντί να σχεδιάσουμε μια ακόμη υπολογιστική μέθοδο μεταξύ των αρκετών ήδη υπάρχοντων εργαλείων πρόγνωσης μεμβρανικών πρωτεΐνών, εστιάζουμε περισσότερο στην αξιολόγηση και σύγκριση των σύγχρονων μοντέλων βαθιάς μάθησης με τα πιο παραδοσιακά, και στην επίδραση της διανυσματικής αναπαράστασης στην προγνωστική ακρίβεια. Αναζητούμε το βαθμό και τους τρόπους με τους οποίους η σχεδίαση ενός μοντέλου και η επιλογή των πρωτεΐνικών αναπαραστάσεων επηρεάζουν την ορθότητα των προγνώσεων. Το κάνουμε για να βοηθήσουμε τους ερευνητές και επιστήμονες βιοπληροφορικής να καταλάβουν τη λειτουργία των μοντέλων πρόγνωσης μεμβρανικών πρωτεΐνών. Χρησιμοποιώντας οκτώ ξεχωριστά μοντέλα ταξινόμησης και δύο εγγενώς διαφορετικές πρωτεΐνικές αναπαραστάσεις, δείχνουμε ότι σε πολλές περιπτώσεις οι παραδοσιακοί αλγόριθμοι συναγωνίζονται επάξια ακόμη και τους πιο εκλεπτυσμένους σύγχρονους. Δείχνουμε ακόμη ότι μια πιο περίπλοκη, βασισμένη στις φυσικοχημικές αμινοξικές ιδιότητες, πρωτεΐνική κωδικοποίηση δεν διασφαλίζει καλύτερα αποτελέσματα από την απλούστερη δυαδική διανυσματοποίηση. Η σημασία αυτής της μελέτης έγκειται στον εμπλουτισμό των γνώσεων μας γενικά περί του θεωρητικού υποβάθρου της πρόγνωσης πρωτεΐνικών τύπων, με έμφαση στον αλγοριθμικό σχεδιασμό και την πρωτεΐνική διανυσματοποίηση.

2 Abstract

The aim of this thesis is to deploy and develop machine and deep learning algorithms (the essential core of data mining techniques) and to account for their performance on membrane protein classification. Rather than designing a new computational method among the several other existing tools for predicting membrane protein types, we focus more on evaluating and comparing newly deep learning models with more traditional ones and on the effect of vector representation on prediction accuracy. We ask to what extent and in which ways model design and the choice of protein vectorization contribute to correct prediction. We do so to better enable researchers and bioinformaticists to understand the mechanics of membrane protein prediction modeling. Using eight distinct classification models and two inherently different protein encoding representations, we show that in many cases traditional algorithms are on par with the new, more sophisticated ones. We also show that a more complex representation based on physico-chemical aminoacidic attributes does not ensure better results than simple encoding. The significance of this study is that it informs our theoretical understanding of protein type prediction in general by introducing a focus on algorithmic design and protein vectorization.

3 Introduction

The cell is considered the most fundamental structural and functional unit of all living organisms and has been justly characterized as the "building block of life". It is highly organized with many functional units or organelles according to the cellular anatomy. Most of these units are enveloped by one or more membranes, which are the structural basis for many important and indispensable for life biological functions.

Membranes give cells their individuality by separating them from their environment. They also control the flow of information between cells and their environment. They do so by being highly selective permeability barriers rather than impervious walls because they contain specific molecular pumps and gates. Because every biological membrane has the same basic bilayer structure, responsible for their distinctive activities are the proteins associated with a particular membrane. Molecules ,for example, can be transported into and out of cells through by membrane proteins such as ion pumps, carrier proteins and channel proteins. The majority of chemical messages such as nerve impulses and hormone activity can be communicated between cells because of membrane proteins. Cytoskeleton parts attach to cell membrane to provide shape with the aid of membrane proteins. Finally, cells grouping together in an extracellular matrix to form tissues, metabolic process completion and defense mechanism success are all thanks to membrane proteins.

The lipid bilayer provides an exceptional and unique hydrophobic environment for membrane proteins. There are proteins buried within the lipid-rich bilayer; other proteins are associated with the exoplasmic or cytosolic leaflet of the bilayer. Protein domains on the extracellular surface of the plasma membrane generally bind to other molecules, including external signaling proteins, ions, and small metabolites (e.g., glucose, fatty acids), and to adhesion molecules on other cells or in the external environment. Domains within the plasma membrane, particularly those that form channels and pores, move molecules in and out of cells. Domains lying along the cytosolic face of the plasma membrane have a wide range of functions, from anchoring cytoskeletal proteins to the membrane to triggering intracellular signaling pathways. In many cases, the function of a membrane protein and the topology of its polypeptide chain in the membrane can be predicted on the basis of its homology with another, well characterized protein[1].

Membrane proteins can be classified based on the nature of the membrane–protein interactions into three major categories: integral single spanning which can be further distinguished based on their orientation in the bilayer in single pass type I, II, III, IV and multipass, lipid-anchored (including GPI-anchored), and peripheral *(Figure 1)*[2].

- Integral membrane proteins, also called transmembrane proteins, span a phospholipid bilayer and are built of three segments. The cytosolic and exoplasmic domains have hydrophilic exterior surfaces that interact with the aqueous solutions on the cytosolic and exoplasmic faces of the membrane. These domains resemble other water-soluble proteins in their aminoacid composition and structure. In contrast, the 3-nm-thick membrane-spanning domain contains many hydrophobic aminoacids whose side chains protrude outward and interact with the hydrocarbon core of the phospholipid bilayer. In all transmembrane proteins examined to date, the membrane-spanning domains consist of one or more α helices or of multiple β strands. In addition, most transmembrane proteins are glycosylated with a complex branched sugar group attached to one or several aminoacid side chains. Invariably these sugar chains are localized to the exoplasmic domains[1].

- Lipid-anchored membrane proteins are bound covalently to one or more lipid molecules. The hydrophobic carbon chain of the attached lipid is embedded in one leaflet of the membrane and anchors the protein to the membrane. The polypeptide chain itself does not enter the phospholipid bilayer. Peripheral membrane proteins do not interact with the hydrophobic core of the phospholipid bilayer. Instead they are usually bound to the membrane indirectly by interactions with integral membrane proteins or directly by interactions with lipid head groups[1].

- Peripheral proteins are localized to either the cytosolic or the exoplasmic face of the plasma membrane. In addition to these proteins, which are closely associated with the bilayer, cytoskeletal filaments are more loosely associated with the cytosolic face, usually through one or more peripheral (adapter) proteins. Such associations with the cytoskeleton provide support for various cellular membranes; they also play a role in the two-way communication between the cell interior and the cell exterior. Finally, peripheral proteins on the outer surface of the plasma membrane and the exoplasmic domains of integral membrane proteins are often attached to components of the extracellular matrix or to the cell wall surrounding bacterial and plant cells[1].

The function of a membrane protein is highly correlated to the type it belongs to. For instance, transmembrane proteins can function on both sides of membrane or transport molecules across it, while on the other hand, proteins that function on only one side of the bilayer are often associated and interact exclusively with either the lipid monolayer or a protein domain on the particular side. That being said, information about membrane protein type can provide important clues to help determine function of an uncharacterized membrane protein, possibly offering

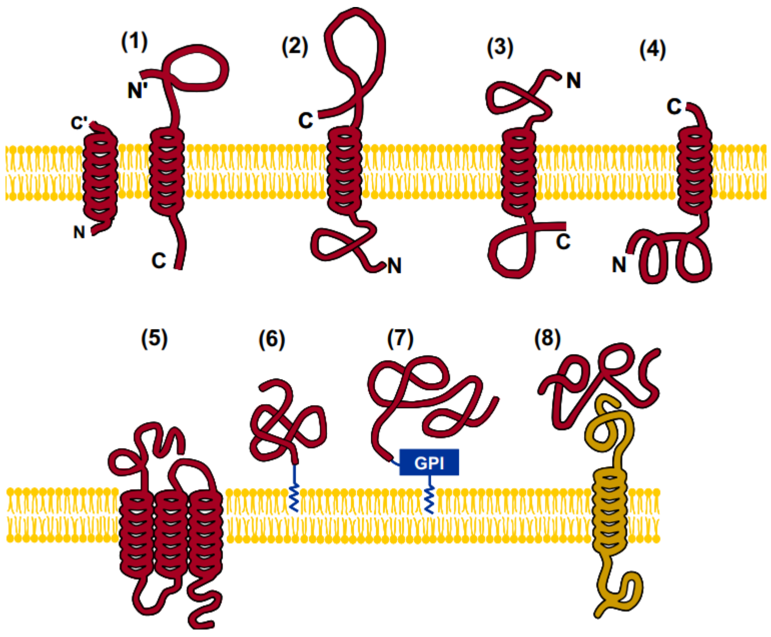


Figure 1: Types of membrane proteins: (1) type I transmembrane, (2) type II, (3) type III, (4) type IV, (5) multipass transmembrane, (6) lipid-chain-anchored membrane, (7) GPI-anchored membrane, and (8) peripheral membrane[3]

key insights to their role in biological processes at the cellular level.

The importance of membrane proteins is suggested from the finding that approximately a third of all yeast genes encode a membrane protein. The relative abundance of genes for membrane proteins is even greater in multicellular organisms in which membrane proteins have additional functions in cell adhesion. The density and complement of proteins associated with biomembranes vary, depending on cell type and subcellular location. For example, the inner mitochondrial membrane is 76% protein; the myelin membrane, only 18% percent[1].

In recent years, the number of sequences entering into protein databases has been rapidly increasing. For instance, the number of total protein sequence entries in UniProt (`https://www.uniprot.org/statistics/Swiss-Prot`) was 563,082 according to version 2020_04 released on 12-Aug-2020, while relative diagrams show a spike in number of entries in the past 15 years *(Figure 2)*. Inspite of the huge increase in protein se-

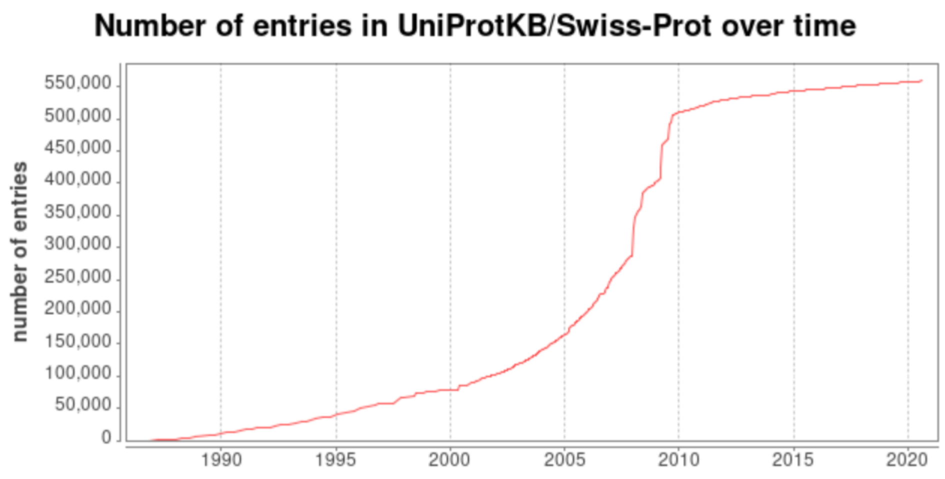


Figure 2: Protein entries in UniProtKB/Swiss-Prot over time

quences entering into databases, membrane proteins still represent $\leq 1\%$ of known protein structures to date[4] even though they are encoded by 20–35% of genes. Even though membrane proteins are elementary and important in nature, determining their 3D structure is proving to be quite a challenging task. Experiments have proven to be both time consuming and rather expensive. Techniques that have shown remarkable success in globular proteins have failed when applied to membrane proteins. Therefore it is of high importance the development of automated methods for fast and effective identification of newly found proteins. Computational methods provide a reliable alternative to experimental methods and incorporate computational algorithms that complement the available experimental techniques and provide useful insights. As a matter of fact, numerous and quite successful models and tools have been proposed in recent years that rely mostly on sequence based information and complex feature engineering. Indicatively we mention: pseudo-amino acid composition PseAAC[5], dipeptide composition (DipC)[6], tripeptide composition (TipC)[7], position specific scoring matrix (PSSM)[6][8].

The majority of these methods are based on machine learning algorithms, applied statistics and probabilistic techniques. All these are the essential core of a discipline that has emerged in recent years: data mining. Data mining is about solving problems by analyzing data already present in databases. By analyzing we mean finding patterns and extracting potentially useful information from them. Data mining addresses the growing gap between data generation and the benefit we get from it by developing the necessary tools to take advantage of hidden information or make it explicit.

4 Background

4.1 Principles of machine learning

Machine learning is essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions [9]. It arises from the question: rather than human beigns developing data processing rules by hand, could a machine automatically learn those rules by processing data *(Figure 3)*? Machine learning allows us to complete tasks that are too hard to solve with fixed programs. From a scientific scope, developing the understanding of how a machine learns encompasses the understanding that may unlock some of the basic principles that underlie intelligence.

A machine learning algorithm is capable of learning from data. But how does a computer learn? Mitchell [10] provides the definition: *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T, as measured by P, improves with experience E."* The process of learning itself is not the task. Learning is our means of attaining the ability to perform the task [9]. Machine learning problems are most of the times defined in terms of how the machine learning algorithm should

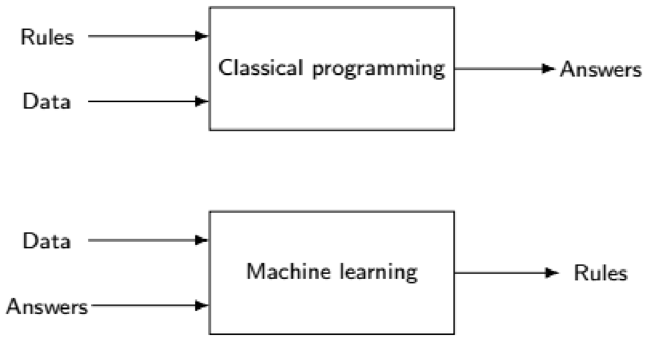


Figure 3: Human learning vs Machine learning

process data. Data are a collection of examples or records each with a set of features that have been quantitatively measured from some object or event that we want the algorithm to process. We typically represent a record as a vector $\boldsymbol{x} \in \mathbb{R}^n$ where each entry x_i of the vector is another feature. Evaluating the abilities of a machine learning algorithm involves using a quantitative measure of its performance. Usually this performance measure P is specific to the task T executed by the algorithm. For tasks such as classification we usually measure the accuracy of the model. Accuracy is the proportion of examples that the model classifies to the correct output and is formally given by:

$$\text{accuracy} = \frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$

We can also obtain equivalent information by measuring the error rate, the proportion of examples for which the model produces an incorrect output. The choice of performance measure may seem straightforward and objective, but it is often difficult to choose a performance measure that corresponds well to the desired behavior of the system [9].

Usually what is of greater interest is how well the algorithm performs on never before seen data, since this determines its performance when deployed in the real world. We therefore evaluate these performance measures using a test set of data that is separate from the data used for training the machine learning system.

Most of the learning algorithms are allowed to experience data as an entire dataset. A dataset is a collection of many examples. Machine learning algorithms are roughly categorized as unsupervised or supervised by what kind of experience they are allowed to have on the dataset during the learning process. Unsupervised learning algorithms are applied to a dataset containing many features, then learn useful properties of the structure of this dataset. Some other unsupervised learning algorithms perform other roles, like clustering, which consists of dividing the dataset into clusters of similar examples. Supervised learning algorithms on the other hand, are fed to a dataset containing features, but each example is also associated with a label or target. A supervised learning algorithm can study the dataset and learn to classify examples to a label based on their features.

Unsupervised learning observes several examples of a random vector $\boldsymbol{x}$, and attempts to implicitly or explicitly learn the probability distribution $p(x)$, or some interesting properties of that distribution, while supervised learning involves observing several examples of a random vector x and an associated value or vector y, and learning to predict y from x, usually by estimating $p(\boldsymbol{y}|\boldsymbol{x})$. The term supervised learning originates from the view of the target y being provided by an instructor or teacher who shows the machine learning system what to do. This is not the case however in unsupervised learning: there is no guidance so the algorithm must learn to make sense of the data on its own.

Unsupervised learning and supervised learning are not formally defined terms. Many machine learning technologies can be used to perform both tasks. The chain rule of probability states that for a vector $x \in \mathbb{R}^n$, the joint distribution can be decomposed as

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i | x_1, ..., x_{i-1})$$

This decomposition means that we can solve the ostensibly unsupervised problem of modeling $p(\boldsymbol{x})$ by splitting it into n supervised learning problems. Alternatively, we can solve the supervised learning problem of learning $p(y|x)$ by using traditional unsupervised learning technologies to learn the joint distribution $p(x, y)$ and inferring

$$p(y|\boldsymbol{x}) = \frac{p(\boldsymbol{x}, y)}{\sum_{y'} p(\boldsymbol{x}, y')}$$

Though unsupervised learning and supervised learning are not completely formal or distinct concepts, they do help to roughly categorize machine learning algorithms. Traditionally, regression, classification and structured output problems are considered supervised learning. Density estimation in support of other tasks is usually considered unsupervised learning.

4.1.1 Generalization

What is of crucial importance in machine learning is *generalization.* Generalization is the ability to perform well on new, unseen data. On known inputs (training set) we determine the error measure (also called the training error) and *optimize* the algorithm by reducing that error. The generalization error, on the other hand, is defined as the expected value of the error on a new input [9] and the aim is to be as low as possible as well.

Affecting performance on unseen data by only observing known inputs can be made possible if some basic assumptions are made. First it is assumed that the examples in each dataset are independent from each other, and secondly that the train set and test set are identically distributed, drawn from the same probability distribution as each other. This assumption allows us to describe the data generating process with a probability distribution over a single example. The same distribution is then used to generate every train example and every test example. We call that shared underlying distribution the data generating distribution, denoted p_{data} [9].

4.1.2 Underfitting and overfitting

Assuming that both datasets are sampled from p_{data} we expect train and test errors to be the same. This is not the case however: because train set sampling preceeds optimization, generalization error is always greater than or equal to the expected train error. Thus a machine learning model must minimize the training error along with its gap to the test error which ultimately boils down to two fundamental challenges: *underfitting* and *overfitting*. High train error results in underfitting whereas overfitting occurs when the gap between the two is large. A model's performance is controlled by its *capacity* or informally how many functions it can fit. This is directly linked to the *hypothesis space*, the set of functions the learning algorithm is allowed to select as the solution [9]. In general models with low capacity are incapable of solving complex tasks while high-capacity ones may have a higher capacity than needed and may overfit.

4.1.3 Hyperparameters and validation set

The model's design must be directly related to the task at hand. It must take into account not only the quantity of allowed functions but also their indentity. Different set of tasks require differrent functions. To control its behavior we do so by tweaking several setting called *hyperparameters*. Hyperparameters differ from the models weights that can be optimized by the model itself and are not adapted by the algorithm. These settings are adjusted by evaluating our model in a held-out test set, composed of examples coming from the same distribution as the training set, called the validation set. A validation set is a convenient way to determine the generalization error before we apply the model to the test set and after learning has completed. When a dataset is relatively small in order to use all examples in the calculation of the mean test error, we can repeat the computation on different randomly chosen subsets of the training sets in a process called k-fold cross validation where the dataset is splitted into k non overlapping subsets.

4.1.4 Point estimation

Calculating estimates of the model's parameters from their true value, one must denote a point estimate of a parameter $\boldsymbol{\theta}$ by $\hat{\boldsymbol{\theta}}$. Let $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ be a set of m independent and identically distributed data points. A point estimator or statistic is any function of the data

$$\hat{\boldsymbol{\theta}}_m = g(\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)})$$

Any function can qualify as an estimator but a good estimator is a function whose output is close to the true underlying θ that generated the training data [9].

4.1.5 Bias

The bias of an estimator is defined as

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}$$

where the expectation is over the data and $\boldsymbol{\theta}$ is the true underlying value of $\boldsymbol{\theta}$ used to define the data generating distribution. An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be unbiased if $\text{bias}(\hat{\boldsymbol{\theta}}_m) = 0$, which implies that $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$. An estimator $(\hat{\boldsymbol{\theta}}_m)$ is said to be asymptotically unbiased if $lim_{m\to\infty}\text{bias}(\hat{\boldsymbol{\theta}}_m) = 0$, which implies that $lim_{m\to\infty}\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$.

4.1.6 Variance

The variance of an estimator is denoted

$$\text{Var}(\hat{\theta})$$

where the random variable is the training set. Alternately, the square root of the variance is called the standard error , denoted $\text{SE}(\hat{\theta})$. Variance is an indicator of how much the estimator varies as a function of the data sample. It is a measure of the expectation of the data's estimation to vary as we resample the dataset. Variance is commonly used when a finite number of samples is used instead of the whole distibution because the estimate of the true underlying parameter remains uncertain. The standard error of the mean is given by

$$\text{SE}(\hat{\mu}_m) = \sqrt{\text{Var}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}$$

where σ^2 is the true variance of the samples.

4.1.7 Mean squared error

Bias determines the expected deviation from the true value of the function or parameter whereas variance provides a measure of the deviation from the expected estimator value of any particular sampling of the data. A model performs best when its overall expected deviation is minimized. The mean squared error (MSE) measures the total expected deviation and is denoted

$$\text{MSE} = \mathbb{E}\left[(\hat{\boldsymbol{\theta}}_m - \boldsymbol{\theta})^2\right] = \text{Bias}(\hat{\boldsymbol{\theta}}_m)^2 + \text{Var}(\hat{\boldsymbol{\theta}}_m)$$

From the above formula it is clear that ideal eastimators are the ones that are capable of keeping both measures in check.

4.1.8 Gradient based optimization

Machine learning models solve tasks by updating estimates of the solution through an iterative process called optimization. Optimization refers to the task of either minimizing or maximizing the *objective function* or *criterion.* In the case of minimization the objective function is called cost function[1].

It is a well known fact that the derivative $f'(x)$ of a function $y = f(x)$ is a insightful way to alter the value of the function by scaling a small change in x to obtain the corresponding change in y.

$$f(x + \epsilon) \approx f(x) + \epsilon f'(x)$$

When the value of x is minimally altered in the opposite direction of the derivative's sign, $f(x)$ is reduced. This is called gradient descent [11]. Ultimately we are interested in finding points where $f(x)$ has the absolute lowest value, that is points where $f'(x) = 0$ or global minima. Optimization may fail to locate the global minimum when a function may have more than one global minima or a number of local minima.

For functions with multidimensional inputs i.e vectors $f : \mathbb{R} \rightarrow \mathbb{R}$ the gradient is used instead of the derivative. The gradient of f is the vector consisting f all the partial derivatives: $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. Element i of

[1]It may also be called loss function or error function

the gradient is the partial derivative with respect to x_i. In this case we are interested in critical points: points where all elements of the gradient equals zero. Minimizing f in this case requires determining the direction in which f decreases fastest. We then make use of the directional derivative in direction $\boldsymbol{u}$ (a unit vector)

$$\min_{\boldsymbol{u}_i \boldsymbol{u}^\top \boldsymbol{u}=1} \boldsymbol{u}^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \min_{\boldsymbol{u}_i \boldsymbol{u}^\top \boldsymbol{u}=1} \|\boldsymbol{u}\|_2 \|\nabla_{\boldsymbol{x}} f(\boldsymbol{x})\|_2 \cos\theta$$

where θ is the angle between $\boldsymbol{u}$ and the gradient. This technique is called gradient descent and reveals a new point: $\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ where ϵ is the learning rate, a positive scalar determining the size of the step.

Large training sets may be suitable for improved generization but increase the computational cost. In most algorithms the cost functions sums up to the per example loss function and so gradient descent requires calculating

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\theta} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta})$$

which is an operation of $O(m)$ computational cost. Stochastic gradient descent which is a different version of gradient descent aims to solve this problem by approximating the gradient by sampling the training set. A minibatch of uniformly drawn records $\mathbb{B} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m')}\}$ is chosen based on which a model may fit an extremely large training set. The gradient's estimate then is formed:

$$\boldsymbol{g} = \frac{1}{m'} \nabla_{\theta} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta})$$

4.2 Supervised learning

The majority of supervised learning algorithms estimate a probability distribution $p(y|x)$. This is accomplished by using the maximum likelihood estimation for the best parameter vector $\boldsymbol{\theta}$ for a parametric family of distributions $p(y|\boldsymbol{x};\boldsymbol{\theta})$

4.2.1 Decision trees

Decision tree learning is a method for approximating discrete-valued target functions. It is a simple yet widely used classification technique and one of the most popular of inductive inference algorithms. It can solve a classification problem by asking a series of carefully crafted questions about the attributes of the test record. Each time an answer is received, a follow-up question is asked until a conclusion is reached about the class label of the record[12]. The series of questions and their possible answers can be organized in the form of a decision tree, which is a hierarchical structure consisting of nodes and directed edges of three types:

- A root node that has no incoming edges and zero or more outgoing edges.
- Internal nodes, each of which has exactly one incoming edge and two or more outgoing edges.
- Leaf or terminal nodes, each of which has exactly one incoming edge and no outgoing edges.

where each leaf node is assigned a class label. The non-terminal nodes contain attribute test conditions to separate records that have different characteristics. The depth of the tree is the length of the longest path from a root to a leaf. Learned trees can also be re-represented as sets of if-then rules to improve human readability.

In principle, there are exponentially many decision trees that can be constructed from a given set of attributes. Designing the optimal tree is computationally infeasible because of the exponential size of the search space. However, a number of efficient algorithms have been developed to induce a reasonably accurate decision tree such as Hunt's algorithm[13] where the tree is grown recursively by partitioning the records into purer subsets.

Splitting measures are defined in terms of class distribution of the records before and after the split. Let $p(i|t)$ denote the fraction of records belonging to class i at a given node t. The measures developed for selecting the best split are often based on the degree of impurity of the child nodes. The smaller the degree of impurity, the more skewed the class distribution. Examples are:

$$\text{Entropy}(t) = -\sum_{i=0}^{c-1} p(i|t) \log_2 p(i|t)$$

$$\text{Gini}(t) = 1 - \sum_{i=0}^{c-1} [p(i|t)]^2$$

$$\text{Classification error}(t) = 1 - \max_i [p(i|t)]$$

where c is the number of classes. To evaluate the performance of a test condition we compare the impurity of the parent node with the impurity of the child node. The gain Δ is the criterion that determines the goodness of spit:

$$\Delta = I(\text{parent}) - \sum_{j=1}^{k} \frac{N(v_j)}{N} I(v_j)$$

where I is the impurity measure of a given node, N is the total number of records at the parent node, k is the number of attribute values, and $N(v_j)$ is the number of records associated with the child node, v_j. Decision tree induction algorithms often choose a test condition that maximizes the gain Δ. Since $I(\text{parent})$ is the same for all test conditions, maximizing the gain is equivalent to minimizing the weighted average impurity measures of the child nodes. Finally, when entropy is used as the impurity measure, the difference in entropy is known as the information gain, Δ_{info}[12].

4.2.2 K-nearest neighbors

The most basic instance-based method is the k-nearest neighbors algorithm. This algorithm assumes all instances correspond to points in the n-dimensional space $\mathbb{R}^n$. The nearest neighbors of an instance are defined in terms of the standard Euclidean distance[2]. More precisely, let an arbitrary instance x be described by the feature vector $(\alpha_1(x), \alpha_2(x), ..., \alpha_n(x))$ where $\alpha_r(x)$ denotes the value of the rth attribute of instance x. Then the distance between two instances x_i and x_j is[10]

$$d(x_i, x_j) = \sum_{r=1}^{n} \alpha_r(x_i) - \alpha_r(x_j)^2$$

In the discrete valued case of the form: $f : \mathbb{R}^n \rightarrow V$ where V is the finite set $\{v_1, ..., v_s\}$ the algorithm approximates a discrete valued target function f with an estimate $\hat{f}$ which is the most common value of f among the k training examples nearest to x. The algorithm can be divided into two parts:

- Training algorithm: store each traiingn example $(x, f(x))$ in the training list

[2]Other measures of similarity maybe the Manhattan distance, Mahalanobis distance, cosine similarity etc.

- Classification algorithm: for a query instance x_q to be classified

  $$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

  where $x_i ... x_k$ denote the k nearest distances to x_q from the list of training examples and $\delta(a, b) = 1$ if $a = b$ or $\delta = 0$ otherwise.

If we choose $k = 1$, then the algorithm assigns to $\hat{f}(x_q)$ the value $f(x_i)$ where x_i is the training instance nearest to x_q. For larger values of k, the algorithm assigns the most common value among the k-nearest training examples[10].

4.2.3 Naive Bayes

The Bayes classifier uses the Bayes theorem to predict the class of an instance as the one that maximizes the posterior probability. The main task is to estimate the joint probability density function for each class, which is modeled via a multivariate normal distribution. The Naive Bayes classifier assumes that attributes are independent, but it is still surprisingly powerful for many applications[14]. The predicted class for $\boldsymbol{x}$ is given as:

$$\hat{y} = \underset{i}{\operatorname{argmax}} \{P(c_i | \boldsymbol{x})\}$$

The Bayes theorem allows us to invert the posterior probability in terms of the likelihood and prior probability, as follows:

$$P(c_i | \boldsymbol{x}) = \frac{P(\boldsymbol{x} | c_i) \cdot P(c_i)}{P(\boldsymbol{x})}$$

where $P(\boldsymbol{x} | c_i)$ is the likelihood, defined as the probability of observing $\boldsymbol{x}$ assuming that the true class is c_i , $P(c_i)$ is the prior probability of class c_i , and $P(\boldsymbol{x})$ is the probability of observing x from any of the k classes, given as:

$$P(\boldsymbol{x}) = \sum_{j=1}^{k} P(\boldsymbol{x}|c_j) \cdot P(c_j)$$

Since $P(\boldsymbol{x})$ is fixed for a given point we can deduct:

$$\hat{y} = \underset{i}{\operatorname{argmax}} \left\{ \frac{P(\boldsymbol{x}|c_i) \cdot P(c_i)}{P(\boldsymbol{x})} \right\} = \underset{i}{\operatorname{argmax}} \{P(\boldsymbol{x}|c_i) \cdot P(c_i)\}$$

meaning that the the predicted class essentially depends on the likelihood of that class taking its prior probability into account[14].

The likelihood and prior probabilities must be calculated directly from the training dataset $\boldsymbol{D}$. Let $\boldsymbol{D}_i$ be the subset of points in $\boldsymbol{D}$ that are labeled with class c_i.

$$\boldsymbol{D}_i = \{\boldsymbol{x}_j \in \boldsymbol{D} | \boldsymbol{x}_j \text{has class } y_i = c_i\}$$

then the the prior probability for class c_i is estimated as:

$$\hat{P}(c_i) = \frac{n_i}{n}$$

where $n = |\boldsymbol{D}|$ is the size of the dataset and $n_i = |\boldsymbol{D_i}|$ is the size of each specific subset[14].

To estimate the likelihood $P(\boldsymbol{x}|c_i)$, we have to estimate the joint probability of $\boldsymbol{x}$ across all d dimensions $P(\boldsymbol{x} = (x_1, ..., x_d)|c_i)$.

The Naive Bayes approach makes the simple assumption that all the attributes are independent. The independence assumption immediately implies that the likelihood can be decomposed into a product of dimension-wise probabilities:

$$P(\boldsymbol{x}|c_i) = P(x_1, ..., x_d|c_i) = \prod_{j=1}^{d} P(x_j|c_i)$$

4.2.4 Support vector machines

Support vector machines (SVMs) determine a linear model called the *maximum margin hyperplane*. In a two-class data set problem whose classes are linearly separable i.e the maximum margin hyperplane in instance space classifies correctly all training instances giving the greatest separation between classes *(Figure 4)*. The bigger the separation (or margin) the smaller the generalization error. The relation between classifier margin and generalization error may be given by structure risk minimization (SRM):

$$R \leq R_e + \phi\left(\frac{h}{N}, \frac{\log(\eta)}{N}\right)$$

where R is the generalization error of the classifier, R_e its training error, N the number of training examples and ϕ a monotone increasing function of the capacity h.

The instances that are closest to the maximum margin hyperplane — the ones with minimum distance to it — are called *support vectors* and uniquely define the maximum margin hyperplane. In the binary class case, a hyperplane separating the two classes might be written:

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$$

where $\boldsymbol{w}$ and b are parameters of the model to be learned. If we rescale these parameters so that the two parallel hyperplanes upon which lie the examples closest to the decision boundary can be expressed as follows:

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 1$$

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = -1$$

then the margin of the decision boundary is given by:

$$\boldsymbol{w} \cdot (\boldsymbol{x_1} - \boldsymbol{x_1}) = 2$$

$$\|\boldsymbol{w}\| \times d = 2$$

$$d = \frac{2}{\|\boldsymbol{w}\|}$$

In the above example the training examples closest to the decision

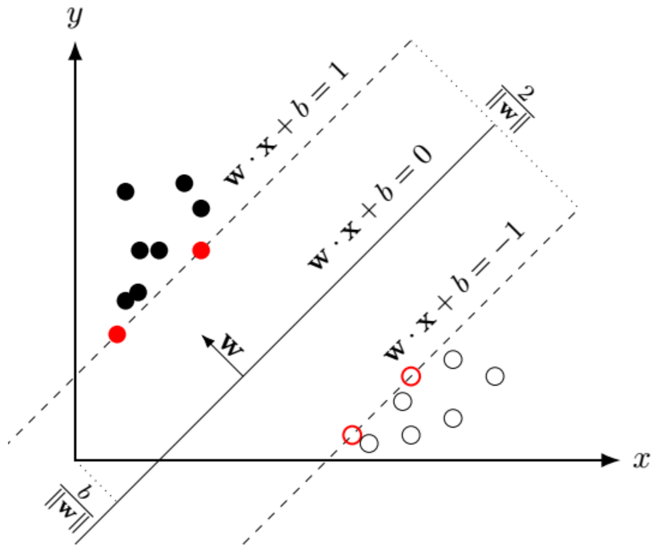

y
x
$\mathbf{w} \cdot \mathbf{x} + b = 1$
$\mathbf{w} \cdot \mathbf{x} + b = 0$
$\mathbf{w} \cdot \mathbf{x} + b = -1$
$\mathbf{w}$
$\frac{2}{\|\mathbf{w}\|}$
$\frac{b}{\|\mathbf{w}\|}$

Figure 4: Maximum margin hyperplane

boundary are called *support vectors.* Determining the support vectors for training instances and thus calculating the parameters b and $\boldsymbol{w}$ falls under a class of optimization problems called *constrained quadratic optimization* problems [15].

Linear models' biggest disadvantage is that they can only represent linear boundaries between classes, which makes them too simple for many practical applications. SVMs use linear models to implement nonlinear class boundaries. These algorithms transform the input using nonlinear mapping. A linear model constructed in the new space can represent a nonlinear decision boundary in the original space [15]. After the transformation the previous methodology is applied to find a linear decision boundary.

This method's biggest drawbacks are computational complexity and overfitting both of which rise due to the large number of the model's coefficients. Overfitting is caused by too much flexibility in the decision boundary. Because the maximum margin hyperplane depends only on support vectors it is relatively stable. This is true even in the high-dimensional space spanned by the nonlinear transformation. On the other hand complexity builds not only for every test instance classified but also during training. To address the problem SVMs introduce a method known as kernel transformation.

A function $(\boldsymbol{x} \cdot \boldsymbol{y})^n$ is called a *polynomial kernel.* In practice, any function $K(\boldsymbol{x} \cdot \boldsymbol{y})$ is a kernel function if it can be written $K(\boldsymbol{x} \cdot \boldsymbol{y}) = \Phi(\boldsymbol{x}) \cdot \Phi(\boldsymbol{y})$, where Φ is a function that maps an instance into a feature space. In other words, the kernel function represents a dot product in the feature space created by Φ. Different kernel functions can be used to implement different nonlinear mappings such as the *sigmoid kernel* and the *radial basis function kernel.* Note that kernel functions are implemented by neural networks that wil be discussed later in full.

The SVM learning problem can be formulated as a convex optimization problem, in which efficient algorithms are available to find the global minimum of the objective function. Other classification methods, such as rule-based classifiers and artificial neural networks, employ a greedy based strategy to search the hypothesis space. Such methods tend to find only locally optimum solutions[12].

4.2.5 Deep feedforward networks

Deep feedforward networks or multilayered perceptrons (MLPs) are supervised learning models where information flows through the function being evaluated from the input $\boldsymbol{x}$, through the intermediate computations used to define f , and finally to the output $\boldsymbol{y}$

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$$

The transformation implemented to data input by a layer is parameterized by its weights $\boldsymbol{\theta}$. In this context, learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets (*Figure 5*).

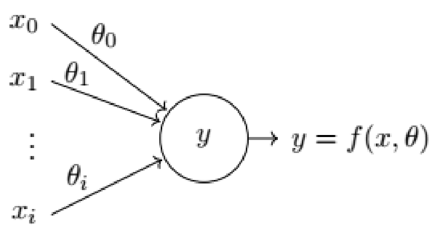


Figure 5: A simple neuron

MLPs put an emphasis on learning successive layers of increasingly meaningful representations. How many layers contribute to a model of the data is called the depth of the model [16]. In deep learning, these layered representations are (almost always) learned via models called neural networks, structured in literal layers stacked on top of each other. These chained structures consist of the first layer (input layer), where data is originally fed to the model, a number of successive layers (specific to each network) where the training data's output is not shown and hence are called hidden layers and finally the final layer called the output layer

(*Figure 6*). Each hidden layer of the network is typically vector-valued.

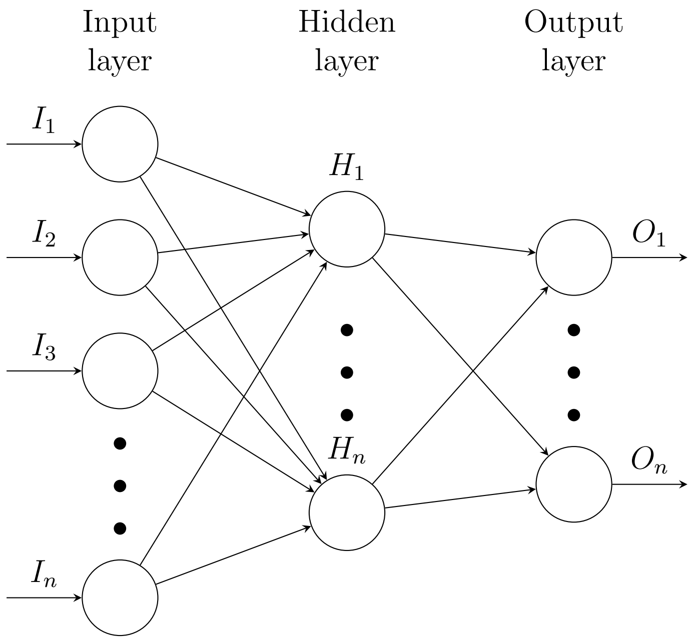


Figure 6: A simple 'neural network'

The dimensionality of these hidden layers determines the width of the model.

In contrast to linear models and to extend them to represent nonlinear functions of $\boldsymbol{x}$, MLPs apply the linear model to a transformed input $\phi(\boldsymbol{x})$ and not to $\boldsymbol{x}$. The ultimate goal is to learn ϕ

$$y = f(\boldsymbol{x}; \boldsymbol{\theta}; \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^\top \boldsymbol{w}$$

where $\boldsymbol{\theta}$ are parameters used to learn ϕ from a broad family of functions and parameters $\boldsymbol{w}$ that map from $\phi(\boldsymbol{x})$ to the desired output.

Crucial to the design of a neural network is the choice of the cost function which is tightly coupled to the choice of the output layer. The choice of how to represent the output then determines the form of the cross-entropy function. In most cases the parametric model defines a distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ and the principle of maximumm likelihood is used

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} \log p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x})$$

The cost function is simply the negative log-likelihood or the cross-entropy between the input and the model's predictions. Furthermore the negative log-likelihood assists in avoiding saturation of the cost's function gradient. In multiclass classification problems the cost function is simply the cross-entropy between the targets y and the probability distribution defined by these unnormalized log probabilities (categorical cross-entropy).

A simpler approach is to merely predict a statistic of $\boldsymbol{y}$ conditioned on $\boldsymbol{x}$. Note that the cost function will often combine a regularization term such as weight decay or dropout layers may be added to the architecture.

4.2.5.1 Hidden units

The design of hidden units is research-active area with no definitive guiding theoretical principles. Determining the correct hidden unit is a process of trial and error: intuiting which hidden unit may perform well, training the model and evaluating its performance on the validation set.

Most hidden units accept a vector of inputs $\boldsymbol{x}$, calculating an affine transformation $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$ and then applying an element-wise non-linear function $g(\boldsymbol{z})$. Rectified linear units[17] (ReLU) are a default choice of hidden unit performing adequately well in most case scenarios. They use the activation function $g(z) = \max\{0, z\}$. ReLU are very similar to linear units with the exception that they output zero across half of their domain. When the unit is active its derivative remains large, plus its gradient is consistent. The fact that their second derivative equals zero across the board makes them an excellent choice for learning. They are usually used on top of the transformation:

$$\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$$

Setting $\boldsymbol{b}$ to relatively small values allows the units to remain active and the derivatives to pass through.

Other commonly used activation functions are the logistic sigmoid: $g(z) = \sigma(z)$ which is an excellent choice as an output unit in binary tasks, and the hyperbolic tangent: $g(z) = \tanh(z)$. Unlike linear units, sigmoidal units saturate to high values when $z \to \infty$ and to low values when $z \to -\infty$. They are only sensitive when $z \sim 0$. This aspect hinders learning, making them a desirable choice in other types of MLPs such as recurrent networks and autoencoders.

If a probability distribution is represented over a discrete variable with n possible values, the softmax function is used. It is a generalization of the sigmoid function used for tasks with n different classes. A vector $\hat{\boldsymbol{y}}$ is required with $\hat{y}_i = P(y = i|\boldsymbol{x})$. Each element of $\hat{y}_i$ must be between 0 and 1, but also $\hat{\boldsymbol{y}}$ must sum to 1 so that it represents a valid probability distribution. A linear layer predicts unnormalized log probabilities

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$$

where $z_i = \log \tilde{P}(y = i|\boldsymbol{x})$. Formally the softmax is given by

$$\operatorname{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum\limits_j \exp(z_j)}$$

4.2.5.2 Regularization

Regularization is a technique especially designed to reduce test error. In general a model is regularized by adding a penalty called regularizer to the cost function. *Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error*[9]. There are many regularization strategies available some of which are:

- *Parameter Norm Penalties*: adds a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function limiting the capacity of the model.
- *Weight decay (ridge regression)*: adds a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ to the objective function driving the weights closer to the origin[3].
- *Dataset Augmentation*: fabricating fake data in order to enlarge our training set and achieve improved generalization.
- *Noise robustness*: adds noise of infinitesimal variance to the inputs, which is equivalent to penalizing weights' norms[18] or to the weights, a strategy often used in recurrent neural networks[19][20].
- *Early stopping*: terminates training when the training error decreases over time but validation error begins to rize again. The models parameters are saved on each cycle and a copy of the last best set is restored rather than the latest one.
- *Dropout*[21]: randomly *dropping out*(setting to zero) a number of output features of a hidden layer during training. Specifically, dropout trains an ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base

[3]More generally, we could regularize the parameters to be near any specific point in space and, surprisingly, still get a regularization effect, but better results will be obtained for a value closer to the true one, with zero being a default value that makes sense when we do not know if the correct value should be positive or negative. Since it is far more common to regularize the model parameters towards zero, we will focus on this special case in our exposition[9].

network. In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero[9].

4.2.5.3 Optimizer

Optimization algorithms used in deep models differ from the ones used in traditional models. Machine learning usually acts indirectly. In most machine learning tasks, we define a performance measure P with respect to the test set and may also be intractable. Thus P is optimized indirectly. A different cost function $J(\theta)$ is reduced in hope to improve P. This differs from pure optimization, where minimizing J is a goal in and of itself. Optimization algorithms for training deep models also typically include some specialization on the specific structure of machine learning objective functions.

The RMSProp algorithm [22] performs better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average. RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl[9]. Empirically, it has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.

4.2.6 Convolutional neural networks

Convolutional neural networks (CNN)[23] are neural networks specialized in grid-like topology data i.e images that represented as 2D grid of pixels. CNNs use a different version of a linear mathematical operation called convolution. *Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of* *their layers.*

In general convolution is an operation on two functions of a real valued argument. It is typically denoted with an asterisk

$$s(t) = \int (x * w)(t)$$

where x is the input function and w, also known as the kernel (or filter), needs to be a valid density function in order for the output to be a valid weighted average of x over time t. The output is reffered as the feature map. In CNNs, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm known as tensors. These functions are assumed equal to zero everywhere but the finite set of points for which we store the values[9].

A typical layer of a CNN consists of three stages. First, the layer performs several convolutions in parallel to produce a set of linear activations. After that, each linear activation is run through a nonlinear activation function, such as ReLU. Lastly, a pooling function is applied to modify the output of the layer further[9].

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. In its essence it is a downsampling technique that reduces computational and statistical burden. Popular pooling functions are max pooling[24] which reports the maximum output within a rectangular neighborhood, the average of the rectangular neighborhood, or a weighted average based on the central output. Pooling helps to make the representation become approximately invariant to small translations of the input. *Invariance to translation means that if we translate the input by a small amount, the values of* *most of the pooled outputs do not change*[9]. Convolution and pooling give CNNs two key characteristics:

- Learned patterns are translation invariant. This means that a after learning a certain pattern, a CNN can recognize it anywhere. A densely connected network would have to learn the pattern anew if it appeared at a new location. This makes convnets data efficient: they need fewer training samples to learn representations that have generalization power[16].

- They learn spatial hierarchies of patterns. A first convolution layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of the features of the first layers, and so on. This allows convnets to efficiently learn increasingly complex and abstract visual concepts[16].

In general, a densely connected layer learns global patterns in its input feature space whereas a convolution layer learns local patterns located in small sized windows.

4.2.7 Wavenets

Wavenets[25] are fully probabilistic and autoregressive neural networks with architectures based on dilated causal convolutions, which exhibit very large receptive fields[25]. In essence Wavenets are stacked convolutional layers, that increase their dilation rate at every layer *(Figure 7)*. This way lower layers learn short-term patterns while higher levels learn long-term patterns. Increasing the dilation rate makes the network capable of processing large sequences very efficiently.

Wavenets have proven to be extremely powerful models that have

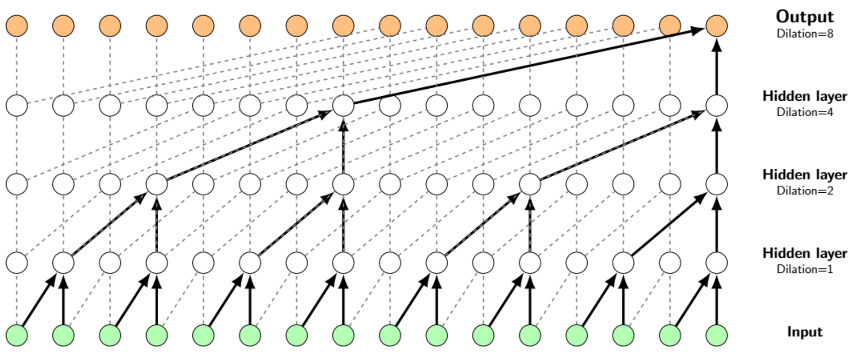


Figure 7: Dilated convolutional layers

outperformed in terms of speed and efficiency other sequence-based architectures by using significantly fewer paramaters.

4.2.8 Recurrent neural networks

Recurrent neural networks or RNNs[26] specialize in processing sequences of values $\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(\tau)}$. RNNs share parameters across diferrent parts of the model. Every part of the output is a function of previous parts of the output. This results in parameter sharing through a deep computational graph.

A computational graph maps inputs and parameters to output and loss. A dynamical system can be described by the recurrent equation:

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta})$$

where $\boldsymbol{s}^{(t)}$ is called the state of the system which at time t refers back to the state at time $t-1$. For a number of time steps τ the graph can be unfolded $\tau - 1$ times in a traditional directed acyclic computational graph where no recurrence is involved.

For a RNN with an external signal $\boldsymbol{x}^{(t)}$ the above equation becomes:

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

where the state now is the hidden units of the network and the equation includes information about the whole sequence. In this way the model is trained to predict the future from the past by learning state $\boldsymbol{h}^{(t)}$ as a sum of the previous sequence of inputs up to t.

4.2.9 Long Short-Term Memory RNNs

Long Short-Term Memory networks (LSTM)[27] allow the network to accumulate information over a long duration. In other words LSTMs are networks with 'memory' which they decide to use if it useful or discard.

To achieve this LSTMs introduce self-loops to produce paths where the gradient can flow for long durations[27]. The weight on this self-loop is conditioned on the context, rather than fixed[28]. By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically. In this case, we mean that even for an LSTM with fixed parameters, the time scale of integration can change based on the input sequence, because the time constants are output by the model itself[9].

LSTM cells do not simply apply nonlinearity to the affine transformation of inputs and recurrent units, but also include an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_i^{(t)}$ that has a linear self-loop *(Figure 8)*. However, here, the self-loop weight is controlled by a forget gate unit $f_i^{(t)}$ that sets this weight to a value between 0 and 1 via a sigmoid unit:

$$f_i^{(t)} = \sigma \left(b_i^t + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} \right)$$

where $\boldsymbol{x}^{(t)}$ is the current input vector and $\boldsymbol{h}^{(t-1)}$ is the current hidden layer vector, containing the outputs of all the LSTM cells, and $\boldsymbol{b}^f$, $\boldsymbol{U}^f$, $\boldsymbol{W}^f$ are respectively biases, input weights and recurrent weights for the forget gates[9]. The LSTM state is thus updated with a conditional self

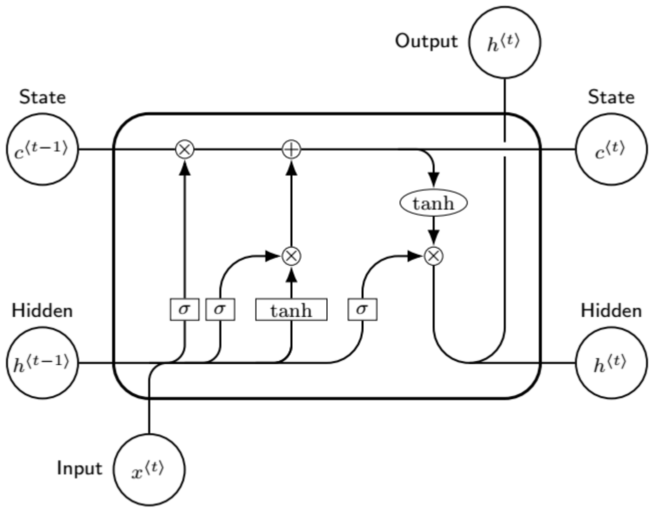


Figure 8: LSTM cell

loop weight $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma \left(b_i^t + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} \right)$$

The external input gate unit $g_i^{(t)}$ is computed similarly to the forget gate:

$$g_i^{(t)} = \sigma \left(b_i^t + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} \right)$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off, via the output gate $q_i^{(t)}$, which also uses a sigmoid unit for gating:

$$h_i^{(t)} = \tanh \left(s_i^{(t)} \right) q_i^{(t)}$$

which has its own parameters (bias, input and recurrent weights), similar to the other gates. LSTMs are capable of learning long term dependencies more efficiently than ordinary RNN architectures especially on challenging sequence processing tasks[29][30][31].

4.3 Unsupervised learning

4.3.1 Principal component analysis

Principal component analysis (PCA) is technique for dimensionality reduction which is a way of compressing data. It is widely used to reduce calculation complexity at the cost of information. In a broader term PCA is an unsupervised learning algorithm which learns a representation of the data with lower dimensionality than the original input. In the new representation the elements have no correlation between each other, hence they are statistically independent.

To achieve this, data are in a sense reconstructed. They are orthogonally and linearly transformed and projected to a new representation with a lower mean squared error. The tradeoff here is the loss of information, so one goal is to preserve as much as possible.

Let $\boldsymbol{X}$ be a $m \times n$ dimensional matrix. Assuming that the data has a mean of zero, $\mathbb{E}[\boldsymbol{x}] = 0$[4]. The unbiased sample covariance matrix associated with $\boldsymbol{X}$ is given by

$$\mathrm{Var}[\boldsymbol{x}] = \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X}$$

PCA linearly transforms the representation to a new one: $\boldsymbol{z} = \boldsymbol{x}^\top \boldsymbol{W}$ where $\mathrm{Var}[\boldsymbol{z}]$ is diagonal. The principal components of $\boldsymbol{X}$ are given by the eigenvectors of $\boldsymbol{X}^\top \boldsymbol{X}$ so

$$\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{W}\boldsymbol{\Lambda}\boldsymbol{W}^\top$$

[4]If this is not the case, the data can easily be centered by subtracting the mean from all examples in a preprocessing step.

When the data $\boldsymbol{x}$ is projected to $\boldsymbol{z}$ via the linear transformation $\boldsymbol{W}$ the resulting representation has a diagonal covariance matrix which ultimately explains why the individual elements of $\boldsymbol{z}$ are mutually uncorrelated.

The transformed representation attempts to disentangle the unknown factors of variation underlying the data [9]. In a more intuitive way, PCA is merely a form of calculating a rotation of the input space (described by $\boldsymbol{W}$) that aligns the principal axes of variance with the basis of the new representation space associated with $\boldsymbol{z}$.

5 Materials and Methods

5.1 Datasets

The datasets used for deploying the algorithms and evaluating our models were benchmark datasets for identifying multi-functional types of membrane proteins accessible at `http://bioinfo.eie.polyu.edu.hk/MemADSVMServer/datasets.html` [32].

The training set, Dataset I, contains 5,307 membrane proteins which were extracted from `https://www.uniprot.org/`[5] released in March 2013[33]. The proteins are divided into eight different functional types: (1) single-pass type I; (2) single-pass type II; (3) single-pass type III; (4) single-pass type IV; (5) multi-pass; (6) lipid-anchor; (7) GPI-anchor and (8) peripheral. Of the total 5,307 proteins 5,117 proteins belong to one functional type, 185 to two types, 5 to three types. The pair-wise sequence identity of the proteins was cut off at 25%.

The test set, Dataset II, contains 3,249 membrane proteins extracted from `https://www.uniprot.org/` in October 2006 [3]. Proteins in the

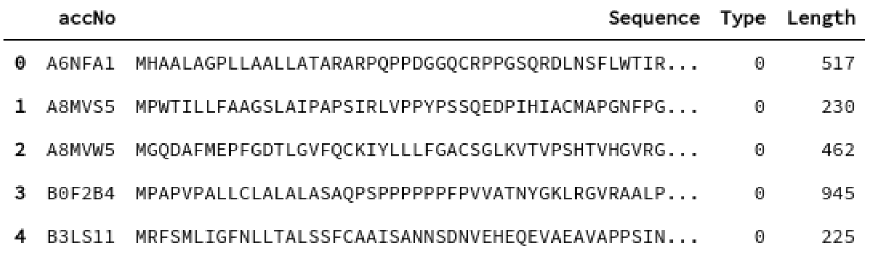
[5]formerly known as Swiss prot

test set are split to the same functional types as the ones in Dataset I, but pair-wise sequence identity of the proteins was cut off at 80% (*Table 1*).

Membrane Protein types	**Train set**	**Test set**
type I	626	610
type II	299	312
type III	42	24
type IV	73	44
Multipass	2,437	1,316
Lipid-chain-anchored	403	151
GPI-anchored	172	182
Peripheral	1,450	610
Total	**5,502**	**3,249**

Table 1: Protein distribution of train and test sets

Initially the files containing the protein datasets were in *pdf* format and had to be converted into *txt* format in order to be parsed. Protein entries were ordered and grouped by functional type. Each entry consisted of accession number, protein facts and fasta sequence. With the help of *Python* language's regular expressions library a custom function was constructed that extracted accession numbers and sequence storing them in a dataframe along with sequence length and functional type (*Figure 9*).



```
   accNo                                          Sequence  Type  Length
0  A6NFA1  MHAALAGPLLAALLATARARPQPPDGGQCRPPGSQRDLNSFLWTIR...     0     517
1  A8MVS5  MPWTILLFAAGSLAIPAPSIRLVPPYPSSQEDPIHIACMAPGNFPG...     0     230
2  A8MVW5  MGQDAFMEPFGDTLGVFQCKIYLLLFGACSGLKVTVPSHTVHGVRG...     0     462
3  B0F2B4  MPAPVPALLCLALALASAQPSPPPPPPFPVVATNYGKLRGVRAALP...     0     945
4  B3LS11  MRFSMLIGFNLLTALSSFCAAISANNSDNVEHEQEVAEAVAPPSIN...     0     225
```

Figure 9: Protein dataframe (first 5 entries)(Dataset I)

For Dataset I, the protein length distribution revealed that mean average protein length is 551 aminoacids and that 96.47% of proteins consist of less than 1,500 aminoacids (*Figure 10*). Respectively for Dataset

II, 97.56% of proteins are under 1,500 aminoacids and show a mean average length of 522 aminoacids (*Figure 11*). The 1,500 mark is of great significance because it will be used as a cutoff: proteins over 1,500 are considered to be outliers and will be disregarded when constructing the numeric tensors fed to the neural networks. This will be discussed in full.

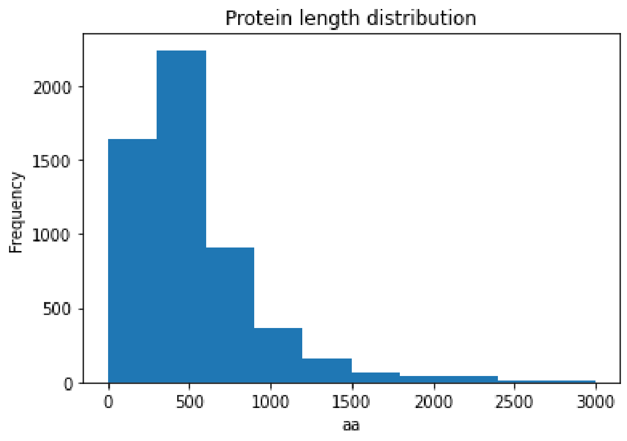


Figure 10: Protein length histogram (Train set)

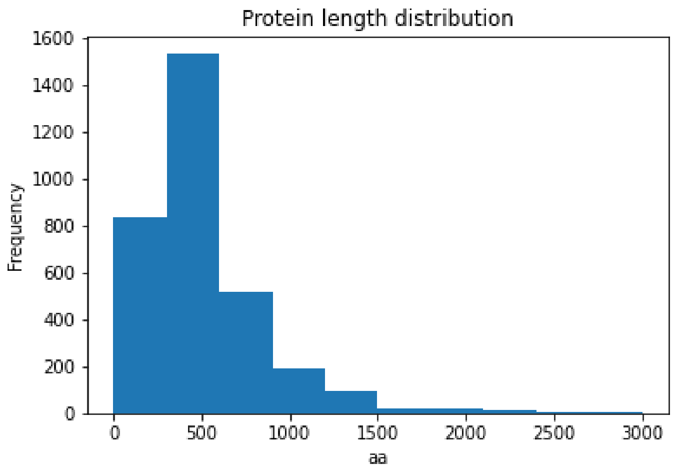


Figure 11: Protein length histogram (Test set)

In general Dataset's I aminoacidic composition depicts the degeneracy of the genetic code (*Figure 12*). Leucine and Serine are in abundance, as one would have suspected since they are the most highly translated aminoacids (six codons each), however this is not the case for the other highly translated aminoacid, Arginine, possibly because of the presence of the charged amino-group in its side chain. Same conclusion can be reached based on Histidine, Cysteine and Tryptophan's low frequency (two codons each).

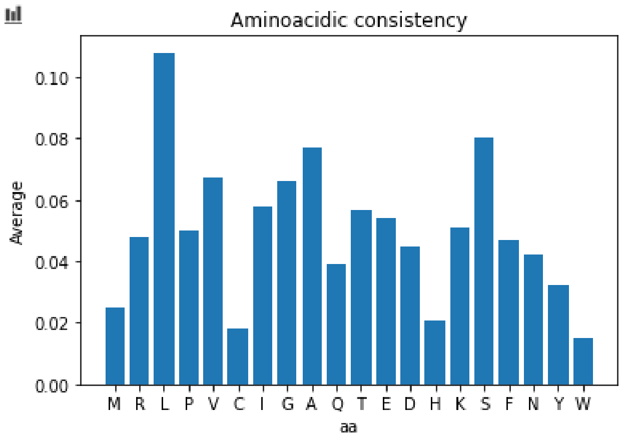


Figure 12: Aminoacidic composition for Dataset I

It should be also noted that aminoacidic content per functional group is characteristic (*Figure 13*). Each group's per aminoacid consistency is generally typical resulting in group distinctiveness. This rules out the posibility of deploying classifiers based on aminoacidic content: classes so well defined would result in near perfect accuracy making it impossible to infer rules for classification or evaluate the models.

Regarding aminoacids, there were 16 proteins in the training set containing one of Z (Glutamic acid or Glutamine), B (Aspartic acid or Asparagine), U (Selenocysteine) or X (any aminoacid) and another 22 in the test set. Retaining these proteins in the datasets, would mean increasing each protein matrix that builds the numeric tensors by four attributes (columns) resulting in a heavy toll in computational cost. Ultimately a decision was made to drop the aforementioned proteins based

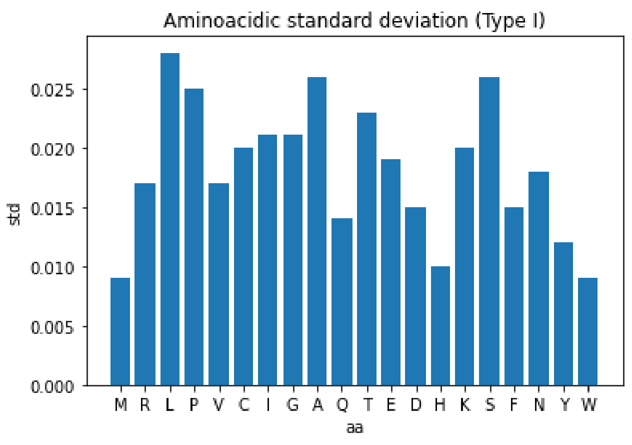


Figure 13: Aminoacidic standard deviation for Type I proteins

on the fact that informational cost would be minimal ($\sim 0.3\%$) compared to the gain in memory resources and computational speed.

Deep-learning models and other classifying algorithms don't take as input raw text or categorical values. Data has to be vectorized. Vectorizing is the process of transforming data into meaningful representantions in the form of numeric tensors. There are quite a few techniques to associate a vector with a sample, *one-hot encoding* being the most widely used.

In one-hot encoding a unique integer index is associated with every value and then this integer index i is turned into a binary vector of size N (the size of the vocabulary which in this case is the 20 aminoacids); the vector is all zeros except for the ith entry, which is 1. In order to create matrices of the same dimensions every protein of the set must be of the same size (contain the same number of aminoacids) so a specific number (cutoff) is chosen. As we have already seen, 96.47% (train set) and 97.56% (test set) of proteins have less than 1,500 aminoacids so sequences over the 1,500 mark were truncated whereas shorter ones were padded with zeros. The vectors are then placed on top of each other represented as a sparse matrix. Ultimately a set of proteins is packed as consecutives matrices in sequence tensors and fed to the network (*Figure 14*).

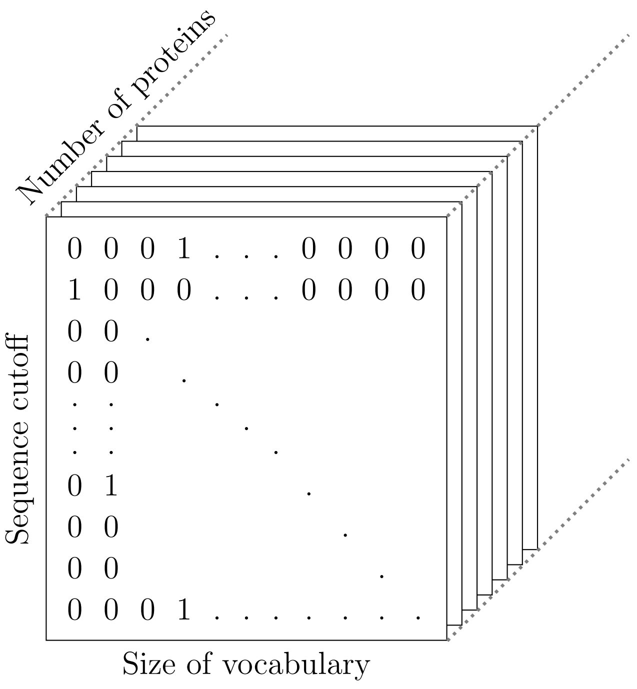


Figure 14: A sequence tensor

One hot encoding although quite effective as a vectorizing technique, remains rather simplistic in its essence. The notion of handling aminoacids independent of each other, ignoring possible relations between them is somewhat 'naive'. Furthermore, models that use sparse matrix representations require abundance in data entries in order to improve performance.

A possibly better representation would be one that incorporates aminoacidic characteristics and their inner relationships such as physicochemical properties as proposed by Guo [34]. To represent aminoacids in this way, AAindex database [35] was used that can be accessed through the DBGET/LinkDB system at GenomeNet (`http://www.genome.ad.jp/dbget/`) or can be downloaded by anonymous FTP (`ftp://ftp.`

`genome.ad.jp/db/genomenet/aaindex/`). AAindex is a database of numerical indices representing various physicochemical and biochemical properties of amino acids. Each property is documented as a set of 20 numerical values (aminoacid index). The AAindex1 section which was used consisted of, at the time this thesis was written, 566 aminoacidic attributes which were parsed with the help of Python language into a dataframe. There were 23 missing values that were replaced with the median value of the particular index.

Using all 566 attributes to describe each aminoacid would greatly increase the size of the protein matrices. Any informational gain from this new vector representation would be counterbalanced by the increase in computational complexity. However AAindex also provides information about highly correlated properties ($\rho \geqslant 0.8$ or $\rho \leqslant -0.8$). Driven by this fact and in order to reduce dimensionality and space complexity, PCA was performed to the attribute dataframe. This resulted in a 18-dimensional space that captured 99.2% of total initial variance. After PCA each aminoacid was assigned its 18-value set and stored in a dictionary for vectorization. The reduced dataframe values varied in different ranges, so in an effort to avoid making the classifiers biased towards larger-valued attributes, they were normalized (mean 0 and standard deviation 1) .

The numeric tensors formed by this alternative approach, differ to the one-hot encoding ones by having two less attributes (dimensions: 1,500 x 18 x dataset size) and are not sparse. This approach reduces computational cost while providing a more meaningful representation.

5.2 Classic machine learning models

Initially we addressed the problem with simpler and more conventional classifiers. Four classic models were put to the test:

- Decision tree with a gini criterion and a depth of 10
- K nearest neighbors using minkowski metric, weights depending on distance and number of neighbors equal to four
- Gaussian Naive Bayes
- Support vector machines with poly kernel, one versus all multiclass strategy, hinge (maximum margin) loss function and dual optimization problem.

Python's language sklearn library (`https://scikit-learn.org/`) was used which is an open source and broadly used tool for classification problems of this type. It should be noted that sklearn classifiers do not work with 3-dimensional tensors but only with 2-dimensional ones. Training and testing sets were flattened out before fed to the models.

5.3 Deep learning models

To test and evaluate the performance of deep learning in membrane protein prediction based on aminoacid sequence, four different models were constructed differing only in design architecture. Hyperparameters such as batch size, activation and loss functions, optimizer and evaluation metrics were kept fixed in order to be able to compare models on the same terms and for the sake of simplicity.

5.3.1 Simple sequential model

The sequential model is the simplest model and consists of two hidden layers of 32 and 16 neurons respectively (*Figure 15*). As input it is fed with the vectorized proteins (1,500x20 matrices) in batches of 64 proteins. Each hidden layer is activated by the ReLU function and ultimately each input is flattened to an 8-dimensional vector (the number of our classes) activated by a softmax function. Rmsprop function is chosen as optimizer and categorical crossentropy as loss function since this is a multiclass classification problem. Lastly accuracy metric is used for evaluation.

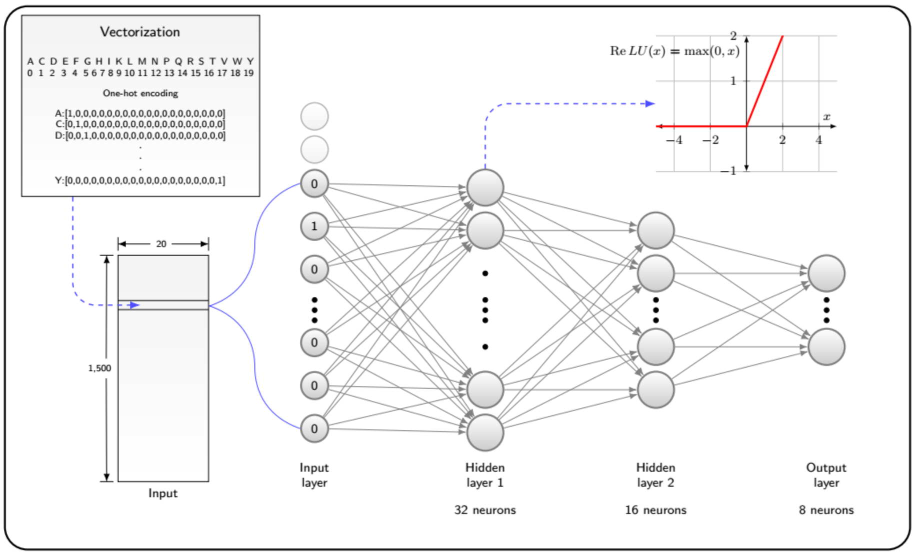

Vectorization
A C D E F G H I K L M N P Q R S T V W Y
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
One-hot encoding
A:[1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
C:[0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
D:[0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Y:[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1]
20
1,500
Input
$ReLU(x) = \max(0, x)$
Input layer
Hidden layer 1
32 neurons
Hidden layer 2
16 neurons
Output layer
8 neurons

Figure 15: Sequential model for one hot encoding approach

5.3.2 Convolutional network

The convolutional model differs from simple sequential models because rather than processing protein sequences as one-dimensional tensors it takes as input whole protein matrices. In essence the proteins matrices are treated as two-dimensional grids of attributes (zeros and ones in the one hot encoding case). There are three convolution layers in the architecture which are all activated by the the ReLU function and ultimately hand their output over to a dense layer of 32 neurons which is furtherly flattened to the 8-dimensional output. The first convolution uses 128 filters and a 15x15 kernel. It is followed by a 64-filter layer with a 5x5 kernel which in turn is followed by the last convolution of 32 filters and 2x2 kernel. Max-Pooling layers that half the output's size are applied in between the convolutions (*Figure 16*). A dropout layer is also added for regularization. Hyperparemeters such as optimizer, loss functions etc. are kept the same as in the sequential model.

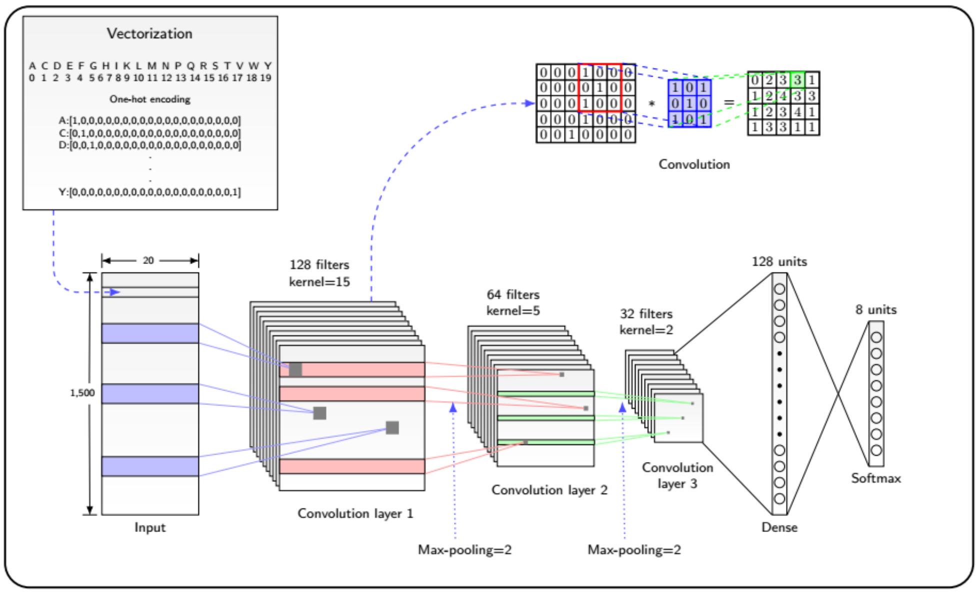


Figure 16: Convolutional model for one hot encoding approach

5.3.3 Long short term memory networks

In an effort to create a more complex architecture which combines convolutions and a recurrent layer, a network with a long short term memory layer was tested. LSTMs are capable of retaining longer memories and thus processing larger sequences and more widely spread apart patterns making them a suitable candidate for these kind of classification problem.

A convolutional layer of 128 filters, 15x15 kernel and a stride of 10 units receives the input and is activated by the ReLU function. A max-pooling layer halves the output which is then handled by a LSTM layer with a 32-dimensional output. The representation is finally passed to the output layer (*Figure 17*). Hyperparameters do not change.

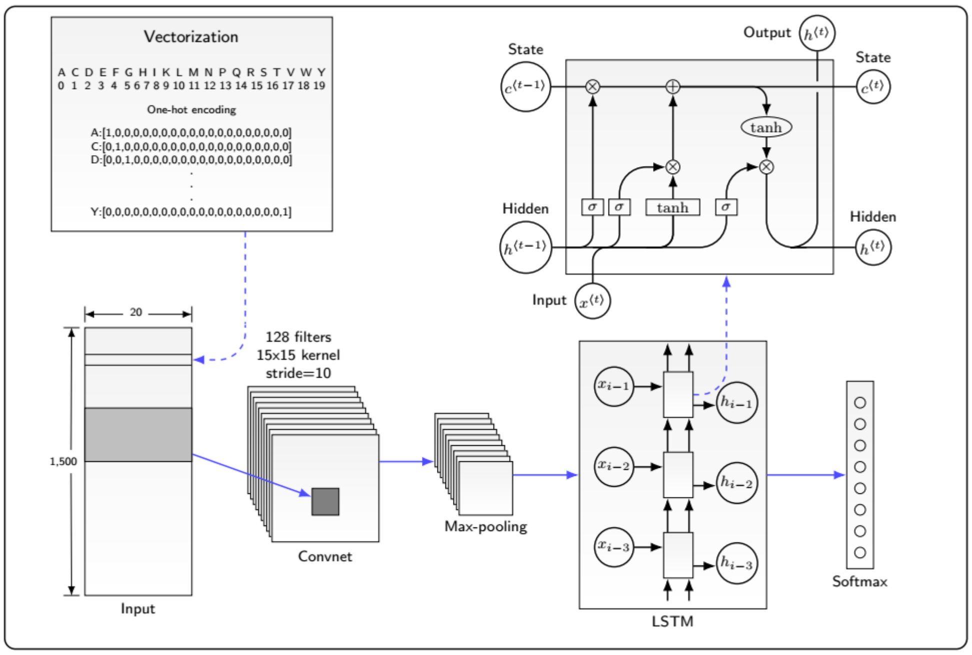

Vectorization
A C D E F G H I K L M N P Q R S T V W Y
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
One-hot encoding
A:[1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
C:[0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
D:[0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Y:[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1]
20
1,500
Input
128 filters
15x15 kernel
stride=10
Convnet
Max-pooling
LSTM
Softmax
State
Output
Hidden
Input
tanh
σ

Figure 17: LSTM model for one hot encoding approach

5.3.4 Wavenets

A wavenet is a variation of a convolutional network that uses dilated causal convolutions. The model consists of two identical convolutional layers with a dilation rate of 2, 4, 8, 16 stacked on top of each other. They use 64 filters and a 15x15 kernel. The architecture leads to one more 64-filter convolution of 2x2 kernel. Before led to the output layer the representation passes through a dropout layer for regularization reasons. Activation function is the ReLU function and hyperparameters are kept same as the other models.

6 Results

As stated before, the metric used for performance evaluation was accuracy. One thing to note is that because of random value assignment to initial weights, every deep learning model's iteration will yield slightly different results. This being the case, every model was deployed five times and accuracy results presented here are the mean average of the five.

It has to be clarified that apart from test set, deep learning models also offer validation set accuracy so this is also included in the results. In the case of validation set, however, a model yields different results with each epoch. After a certain number of epochs though, performance begins to saturate (*Figure 18*). Similarly to test set results, validation set accuracy metric presented here is the mean average of epochs. For the

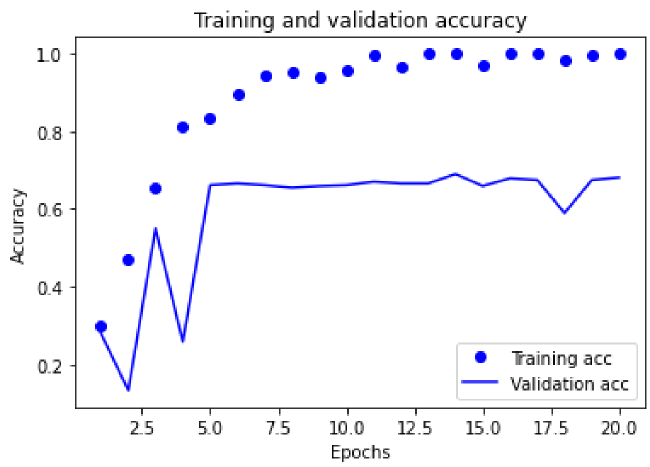


Figure 18: Performance saturation

one hot encoding method the results are presented in the table below:

Accordingly classifier performance for the alternative encoding method using physicochemical aminoacid attributes are presented in the below

One hot encoding

Classifier	Validation set	Test set
Decision tree	-	0.38
Naive bayes	-	0.48
K-nearest neighbors	-	0.65
Support vector machines	-	0.77
Simple Sequential	0.60	0.64
Convnet	0.75	0.78
LSTM	0.68	0.67
Wavenet	0.80	0.84

Table 2: Classifier performance (accuracy)

table: Alternatively we present accuracy metric results (test set) for

Physicochemical attribute encoding

Classifier	Validation set	Test set
Decision tree	-	0.27
Naive bayes	-	0.12
K-nearest neighbors	-	0.65
Support vector machines	-	0.76
Simple Sequential	0.63	0.62
Convnet	0.77	0.75
LSTM	0.61	0.63
Wavenet	0.81	0.82

Table 3: Classifier performance (accuracy)

both encoding methods side by side for each classifier, for easier comparison:

7 Discussion

It has to be stressed out that the aim of this research was to compare techniques and representations and not to develop a new prediction tool. Hyperparameter tweaking and architectural design were not examined to extreme detail and this has to be taken into account in this discussion. Above everything else, it is hard not to observe that this is an unbal-

Classifier performance

Classifier	**One hot enc**	**Phys-chem enc**
Decision tree	0.38	0.27
Naive bayes	0.48	0.12
K-nearest neighbors	0.65	0.65
Support vector machines	0.77	0.76
Simple Sequential	0.64	0.62
Convnet	0.78	0.75
LSTM	0.67	0.63
Wavenet	0.84	0.82

Table 4: Accuracy metric results

anced class set classification problem. There is a dominant class (multipass type proteins 40%) and classes that are less than 1% frequent (type 3 proteins). A number of available techniques that can improve performance have been developed for these kind of problems: weight optimization, sub-sampling and SMOTE [36]. In our case, class unbalance has been addressed by optimizing class weights. This was accomplished by initially feeding to each model's hyperparameters an array of sample weights adjusted inversely proportional to class frequencies. It has to be noted that even though weight optimization was performed results were marginally better ($< 1\%$). It remains to be seen if sub-sampling or SMOTE yield improved results and are left as future work. Although unbalanced, the similarity between train and test set distributions (if not identical) smooths out model performance, this being apparent from the marginal difference between validation and testing set accuracy results. In any case, this was to be expected because that was the case from the beginning. However having this kind of tangible results proves model design wellness and provides hard evidence for correct hyperparameterization.

Regarding classic machine learning models, flattening input protein sequences into a 2D matrix played a significant role in training and has to be taken into account when evaluating their performance. For the one-hot encoding representation, compared to the baseline accuracy, which can be hypothetically (and quite roughly) estimated as a model (baseline) that classifies all proteins in the test set as multipass proteins (most frequent class: 40% of population), the majority of the models scored relatively high precision with the exception of Decision trees. On average, classical methods scored 56% correctly classifying over half of the proteins with Support vector machines yielding a relatively impressive 78%, being on par with the more sophisticated deep learning models. K-nearest neighbors model with its simplistic distance based approach performed equally as well as the LSTM. Decision tree algorithm based model scored the lowest while the Gaussian Naive Bayes one performed marginally better than hypothetical baseline model.

On the other hand, deep learning models correctly classified on average 73.5% of proteins, rendering them an overall more suitable choice of models for these kind of classification problems. The simple sequential model with only two hidden layers of neuron scored the lowest accuracy (64%), while the Wavenet model the highest (84%) and is a force to be reckoned with. In general convolutional models performed better than the LSTM which has the ability to retain 'memories', and was expected to be more suitable for sequence processing.

One has to understand that this is in its core a sequence data problem. It is widely accepted that sequence analysis problems are best addressed by recurrent and convolutional networks because of the former's ability to grow memory and the latter's intrinsic design that allows them to process the input segment by segment instead of it whole. In our case, the models have to process sequences of data understood as sequences of aminoacids. Each protein consists of a very specific and exclusive to each one, aminoacidic sequence, distinguishing one from another. Furthermore, aminoacidic order matters: one misplacement may lead to a completely different protein with potentially catastrophic consequences, so it makes sense that a recurrent model which is able to 'remember' which aminoacid comes after what in a specific domain will perform better than conventional ones. However the results prove that this is not the case. LSTM performed relatively poorly compared to convnets. A possible reason maybe that even though capable of retaining memories there are limitations: it is impossible to remember relatively long sequences. Another reason maybe that our LSTM processes sequences one way. It is a well fact [37] that bidirectional LSTMs are capable of improved performance in genome sequence analysis. Deployment of bidirectional LSTM is left however as future work.

Convolutional networks on the other hand are highly relevant and can perform particularly well on sequence analysis problems because they operate convolutionally, extracting features from local input bands and allowing for representation modularity and data efficiency. In fact convolutional networks can be computationally cheaper but equally competitive with recurrent models making them an excellent choice for sequence processing. In essence, convolutional networks can recognize local patterns in a sequence. The same kernel transformation is performed on each band so a pattern learned at a specific position in the sequence can be identified anywhere else. Aminoacidic sequences such as motifs, supersecondary structures and domains can thus be easily recognized anywhere in the protein. This explains to an extend their overall success in this classification.

Wavenet model's high score is due not only to convolutional network's intrinsic design, but also to the fact that dilated convolutions can process even longer sequences than LSTMs. This is a key feature in sequential data analysis, especially in genome or proteome sequences where a local pattern maybe depended or related to another one hundred or even thousand aminoacids away. Since proteins are produced in aminoacidic order (a cartain aminoacid follows a certain aminoacid), they can be considered as extremely correlated time series data. Due to the importance of order, it can also be thought of as long dependency data so a model needs to consider many prior samples to predict the next value. Wavenets address this by increasing exponentially their receptive field with linearly increasing number of parameters. This is probably one of the reasons for Wavenets' high prediction accuracy.

Of great interest is the effect of vector representation on model performance. From the results it is clear that for the minimalistic binary one-hot encoding vectorization models scored marginally better accuracy. This is apparent in every model. In essence incorporating intrinsic aminoacidic characteristics such as physico-chemical attributes did not improve performance. One might argue that attribute value fluctuation played a role but this is not the case here: input values were normalized before training so input value range does not come into play in the results. It has to be noted that the choice of vector representation is relevant to the problem: one-hot encoding might result in high evaluation metrics in membrane protein classification but may perform poorly in problems of different nature.

8 Conclusion

In this thesis we demonstrated how data mining techniques can be applied to biology oriented prediction problems. More specifically we deployed 8 machine learning methods, that rely on sequence based algorithms in order to classify membrane proteins to various types. Model performance was quite promising, even though this was not the aim of the research, and seem to affirm the common belief that deep learning methods are the go-to method for protein type prediction. Traditional algorithms proved to be quite competitive especially support vector machines, and are a force to be reckoned with. Deep learning algorithms that have the ability to process longer sequences and recognize local patterns seemed to perform the best but future work needs to be done in order to reach more solid conclusions.

Protein vectorization was addresed in two ways: a one-hot encoding representation and one based on physicochemical aminoacidic attributes. Results showed that one-hot encoding, although simplistic, yielded marginally better accuracy than the physico-chemical approach but this is also a matter that needs further investigation.

We showed that the combination of theoretical methods and computational tools can complement, if not replace the more expensive and time consuming experimental ones. It is our belief that the recent breakthroughs in machine learning algorithmic development, along with the ever-increasing amount of experimental data and the rise in computational power will fuel novel techniques and approaches boosting research in the field.

Bibliography

[1] Lodish H. *et al. Molecular Cell Biology 5th ed. (Chapter 5)* W. H. Freeman, New York, 2003

[2] Spiess M. *Heads or tails — what determines the orientation of proteins in the membrane.* FEBS Letters, 369.1 (pages 76-79), 1995

[3] Kuo-Chen Chou, Hong-Bin Shen. *MemType-2L: A Web server for predicting membrane proteins and their types by incorporating evolution information through Pse-PSSM.*, BBRC 360(2) (pages 339-345), 2007

[4] Almeida J. *et al. Membrane proteins structures: A review on computational modeling tools.* Biochimica et Biophysica Acta (BBA), 1859.10 (pages 2021-2039), 2017

[5] Chou KC. *Prediction of protein cellular attributes using pseudo-amino acid composition.* Proteins-Struct Funct Bioinforma. 43(3) (pages 246–255), 2010

[6] Wang T, Xia T, Hu XM *Geometry preserving projections algorithm for predicting membrane protein types.* J Theor Biol. 262(2) (pages 208–213), 2010

[7] Anishetty S, Pennathur G, Anishetty R. *Tripeptide analysis of protein structures.* BMC Struct Biol. 2(1) (pages 9), 2002

[8] Wang T, *et al. Predicting membrane protein types by the LLDA algorithm.* Protein Pept Lett.15(9) (pages 915–921), 2008

[9] Ian Goodfellow, Yoshua Bengio, Aaron Courville. *Deep Learning.* The MIT Press, Cambridge, Massachusetts, London, England, 2016

[10] Mitchell, T. M. *Machine learning.* McGraw-Hill, New York, 2018

[11] Cauchy, A. *Méthode générale pour la résolution de systèmes d'équations simultanées.* Compte rendu des séances de l'académie des sciences (pages 536–538), 1847

[12] Pang-Ning Tan *et al. Introduction to Data Mining.* Pearson Education Limited, Essex, England, 2014

[13] Hunt *et al. Experiments in induction.* New York: Academic, 1966

[14] Zaki M.J. and Meira W. *Data Mining and Analysis: Fundamental Concepts and Algorithms.* Cambridge University Press, New York, 2014

[15] Ian H. Witten *et al. Data Mining Practical Machine Learning Tools and Techniques.* Elsevier, Cambridge, Massachusetts, United States, 2017

[16] Francois Chollet. *Deep Learning with Python.* Manning Publications Co, Shelter Island, NY, United States, 2018

[17] Nair, Vinod and Hinton, Geoffrey E. *Rectified linear units improve restricted Boltzmann machines.* ICML, (pages 807-814), 2010

[18] Bishop, C. M. *Training with noise is equivalent to Tikhonov regularization.* Neural Computation, 7(1) (pages 108–116), 1995

[19] Jim *et al. An analysis of noise in recurrent neural networks: convergence and generalization. IEEE Transactions on Neural Networks*, 7(6) (pages 1424-1438), 1996

[20] Graves, A. *Practical variational inference for neural networks.* NIPS, 2011

[21] Srivastava *et al. Dropout: A simple way to prevent neural networks from overfitting.* Journal of Machine Learning Research, 15 (pages 1929–1958), 2011

[22] Hinton, G. *et al. Improving neural networks by preventing co-adaptation of feature detectors.*, Technical report, arXiv:1207.0580. (pages 238, 263, 267), 2012

[23] LeCun, Y. *Generalization and network design strategies.* Technical Report, CRG-TR-89-4, University of Toronto, 1989

[24] Zhou, Y. and Chellappa, R. *Computation of optical flow using a neural network.* In Neural Networks, IEEE International Conference (pages 71–78), 1988

[25] Aaron van den Oord *et al. WaveNet: A Generative Model for Raw Audio.* arXiv preprint arXiv:1609.03499, 2016

[26] Rumelhart, D.E. *et al. Learning representations by back-propagating errors.* Nature 323 (pages 533-536), 1986

[27] Hochreiter, S. and Schmidhuber, J. *Long short-term memory.* Neural Computation, 9(8) (pages 1735–1780), 1997

[28] Gers, F.A. *et al. Learning to forget: Continual prediction with LSTM.* Neural Computation, 12(10), (pages 2451–2471), 2000

[29] Graves, A. *Supervised Sequence Labelling with Recurrent Neural Networks.* Studies in Computational Intelligence. Springer , 2012

[30] Graves, A. *et al. Speech recognition with deep recurrent neural networks.* ICASSP, (pages 6645–6649), 2013

[31] Sutskever, I. *et al. Sequence to sequence learning with neural net-*

works. NIPS, arXiv:1409.3215, 2014

[32] Wan S. *et al. Benchmark data for identifying multi-functional types of membrane proteins.* Data Brief.8(C):105–7, 2016

[33] Xiao X. *et al. iMem-Seq: a multi-label learning classifier for predicting membrane proteins types.*, J. Membr.Biol. 248(4) (pages 745–752), 2015

[34] Lei Guo *et al. Accurate classification of membrane protein types based on sequence and evolutionary information using deep learning.*, BMC Bioinformatics 20(700), 2019

[35] Kawashima S. *et al. AAindex: Amino Acid Index Database.*, Nucleic Acids Res 27(1) (pages 368–369), 1999

[36] Chawla N. *et al SMOTE: Synthetic Minority Over-sampling Technique.*, JAIR 16. (pages 321-357), 2002

[37] N. Tavakoli *Modeling Genome Data Using Bidirectional LSTM.*, IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC) (pages 183-188), 2019

Appendices

Appendix A Python code

```python
import re
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from keras.utils.np_utils import to_categorical
from keras import models
from keras.layers import Flatten
from keras import layers
from sklearn import preprocessing
from sklearn.decomposition import PCA
from sklearn.metrics import precision_recall_fscore_support
from sklearn import metrics
from sklearn.naive_bayes import GaussianNB
from sklearn import tree
from sklearn import svm
from sklearn.neighbors import KNeighborsClassifier


symbol = ['M','R','L','P','V','C','I','G','A','Q','T','E','D','H','K','S','F','N','Y','W']

fastaTrain = open(r"C:\pythonProjects\thesis\Supp-S1.txt", "r")
fastaTest = open(r"C:\pythonProjects\thesis\Supp-A.txt", "r")
attributes = open(r"C:\pythonProjects\thesis\aaindex1.txt", "r")

def getDataFromTrainFile (file):
    seq=''
    accNumber=[]
    sequences=[]
    classes=[]
    dataList = [accNumber,sequences,classes]
    for line in file:
        line=line.strip()
        x=re.search('^>',line)
        y=re.search('^[A-Z]+$',line)
        z=re.search('^\(\d',line)
        if z!=None:
            seqClass=re.findall('^\(',line)
            dataList[2].append(line)
        elif x!=None:
            accNo=re.search('\_\S{6}',line)
            dataList[0].append(accNo.group()[1:])
            if seq!='':
                dataList[1].append(seq)
                seq=''
        elif y!=None:
            seq=seq+y.string
    dataList[1].append(seq)
    return dataList

    def getDataFromTestFile (file):
    seq=''
    accNumber=[]
    sequences=[]
    classes=[]
    dataList = [accNumber,sequences,classes]
    for line in file:
        line=line.strip()
        x=re.search('^>',line)
        y=re.search('[A-Z]{1,60}',line)
        z=re.search('^\(',line)
        if z!=None:
            seqClass=re.findall('^\(',line)
            dataList[2].append(line)
        elif x!=None:
            accNo=re.search('[^>]{6}',line)
            dataList[0].append(accNo.group())
            if seq!='':
                dataList[1].append(seq)
                seq=''
        elif y!=None:
            seq=seq+y.string
    dataList[1].append(seq)
    return dataList

trainDataset = getDataFromTrainFile(fastaTrain)
testDataset = getDataFromTestFile(fastaTest)

aminoAttr=[]
cnt=0
for line in attributes:
    line=line.strip()
    x=re.search('^-?\d+\.\d*',line)
    if x!=None:
        cnt+=1
        attribs=re.findall('-?\d+\.\d*',line)
        aminoAttr.append(attribs)

floatList=[]
tempList=[]
for i,item in enumerate(aminoAttr):
   for j in item:
      tempList.append(float(j))
   if i%2!=0:
      floatList.append(tempList)
      tempList=[]

myDf = pd.DataFrame(floatList).transpose()
rows,cols = myDf.shape
```

```
for i in range(cols):
    myDf[i].fillna(myDf[i].median(),inplace=True)

pca = PCA(n_components=18)
reducedData = pca.fit_transform(myDf)
reducedData = preprocessing.scale(reducedData)

fastaTest.close
fastaTrain.close
attributes.close

a=np.zeros(626)
b=np.ones(299)*1
c=np.ones(42)*2
d=np.ones(73)*3
e=np.ones(2437)*4
f=np.ones(403)*5
g=np.ones(172)*6
h=np.ones(1450)*7
classes = np.concatenate([a,b,c,d,e,f,g,h],axis=0)

trainDf = pd.DataFrame(zip(trainDataset[0],trainDataset
    [1],classes),columns=['accNo','Sequence','Type'])

a=np.zeros(610)
b=np.ones(312)*1
c=np.ones(24)*2
d=np.ones(44)*3
e=np.ones(1316)*4
f=np.ones(151)*5
g=np.ones(182)*6
h=np.ones(610)*7
classes = np.concatenate([a,b,c,d,e,f,g,h],axis=0)

testDf = pd.DataFrame(zip(testDataset[0],testDataset[1],
    classes),columns=['accNo','Sequence','Type'])
```

```

trainDf['Type'] = trainDf['Type'].astype(np.int8,copy=
    False)
testDf['Type'] = testDf['Type'].astype(np.int8,copy=False)

condition1 = trainDf['Length']<=1500
train1500 = trainDf[condition1]
train1500 = train1500.drop(['Length','accNo'],1)
train1500 = train1500.reset_index(drop=True)

condition2 = train1500['Sequence'].apply(lambda x: ('X'
    not in x) and ('B' not in x) and ('Z' not in x) and ('U
    ' not in x))
train1500 = train1500[condition2]
trainRows,trainCols=train1500.shape

condition1 = testDf['Length']<=1500
test1500 = testDf[condition1]
test1500 = test1500.drop(['Length','accNo'],1)
test1500 = test1500.reset_index(drop=True)

condition2 = test1500['Sequence'].apply(lambda x: ('X' not
     in x) and ('B' not in x) and ('Z' not in x))
test1500 = test1500[condition2]
testRows,testCols=test1500.shape

cnt=0
oneHotDic={}
for i,aa in enumerate(train1500.iloc[0,0]):
    if aa not in oneHotDic:
        oneHotDic[aa]=cnt
        cnt+=1

physicoDictionary = {}
for i in range(20):
    physicoDictionary[symbol[i]]=np.around(reducedData[i
    ,:],2)

np.random.seed(5)

trainArray = train1500.to_numpy()
np.random.shuffle(trainArray)
trainSequences = trainArray[:,0]
trainClasses = trainArray[:,1]
trainClasses=trainClasses.astype(np.int8)

testArray = test1500.to_numpy()
np.random.shuffle(testArray)
testSequences = testArray[:,0]
testClasses = testArray[:,1]
testClasses=testClasses.astype(np.int8)

flag=True
for seq in trainSequences:
    trainRow=[0]*30000
    for i,aa in enumerate(seq):
        trainRow[oneHotDic[aa]+i*20] = 1
    if flag:
        trainForML = trainRow
        flag=False
    else:
        trainForML = np.vstack((trainForML,trainRow))

flag=True
for seq in testSequences:
    testRow=[0]*30000
    for i,aa in enumerate(seq):
        testRow[oneHotDic[aa]+i*20] = 1
    if flag:
        testForML = testRow
        flag=False
    else:
        testForML = np.vstack((testForML,testRow))

trainSet = np.zeros([trainSequences.shape[0],1500,len(
    oneHotDic)])
for i,seq in enumerate(trainSequences):
    for j,aa in enumerate(seq):
        trainSet[i,j,oneHotDic[aa]]=1

testSet = np.zeros([testSequences.shape[0],1500,len(
    oneHotDic)])
for i,seq in enumerate(testSequences):
    for j,aa in enumerate(seq):
        testSet[i,j,oneHotDic[aa]]=1

trainSampleweight = class_weight.compute_sample_weight('
    balanced',trainClasses)

clfDT = tree.DecisionTreeClassifier(max_depth=10,
    min_samples_split=2,min_samples_leaf=1, class_weight='
    balanced')
clfNB = GaussianNB()
clfNN = KNeighborsClassifier(n_neighbors=4,weights='
    distance',metric='minkowski')
clfSVM= svm.LinearSVC(class_weight='balanced')

clfDT.fit(trainForML, trainClasses, sample_weight=
    trainSampleweight)
clfNB.fit(trainForML, trainClasses, sample_weight=
    trainSampleweight)
clfNN.fit(trainForML, trainClasses, sample_weight=
    trainSampleweight)
clfSVM.fit(trainForML, trainClasses, sample_weight=
    trainSampleweight)

y_test_pred_DT=clfDT.predict(testForML)
y_test_pred_NB=clfNB.predict(testForML)
```

```
y_test_pred_NN=clfNN.predict(testForML)
y_test_pred_SVM=clfSVM.predict(testForML)

precisionDT=precision_recall_fscore_support(testClasses,
    y_test_pred_DT, average='macro')[0]
precisionNB=precision_recall_fscore_support(testClasses,
    y_test_pred_NB, average='macro')[0]
precisionNN=precision_recall_fscore_support(testClasses,
    y_test_pred_NN, average='macro')[0]
precisionSVM=precision_recall_fscore_support(testClasses,
    y_test_pred_SVM, average='macro')[0]

trainClasses = to_categorical(trainClasses,num_classes=8,
    dtype='int8')
testClasses = to_categorical(testClasses,num_classes=8,
    dtype='int8')

trainSeq = trainSet[:3000,:1500,:20]
valSeq = trainSet[3000:,:1500,:20]
trainClass = trainClasses[:3000,:8]
valClass = trainClasses[3000:,:8]

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape
    =(1500,20)))
model.add(layers.Dense(16, activation='relu'))
model.add(Flatten())
model.add(layers.Dense(8, activation='softmax'))

model = models.Sequential()
model.add(layers.Conv1D(128, (15), activation='relu',
input_shape=(1500,20)))
model.add(layers.MaxPooling1D((2)))
model.add(layers.Conv1D(64, (5), activation='relu'))
model.add(layers.MaxPooling1D((2)))
model.add(layers.Conv1D(32, (2), activation='relu'))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dropout(0.3))
model.add(layers.Flatten())
model.add(layers.Dense(8, activation='softmax'))

model = models.Sequential()
model.add(layers.Conv1D(128, (15),strides=10, activation='
    relu',
input_shape=(1500,20)))
model.add(layers.MaxPooling1D((2)))
model.add(layers.LSTM(64))
model.add(layers.Flatten())
model.add(layers.Dense(8, activation='softmax'))

model = models.Sequential()
model.add(layers.InputLayer(input_shape=(1500,20)))
for rate in (1, 2, 4, 8):
    model.add(layers.Conv1D(64, (15),
    padding="causal",activation="relu", dilation_rate=rate
    ))
model.add(layers.Conv1D(64,(2)))
model.add(layers.Dropout(0.3))
model.add(layers.Flatten())
model.add(layers.Dense(8, activation='softmax'))

model.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])

history = model.fit(trainSeq,trainClass,
epochs=20,
batch_size=64,
sample_weight=trainSampleWeight,
validation_data=(valSeq, valClass))

model.evaluate(testSet,testClasses,batch_size=64,
    sample_weight=testSampleWeight)
```