SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ DI
INGEGNERIA DELL'INFORMAZIONE, INFORMATICA E STATISTICA

DEPARTEMENT OF
COMPUTER, CONTROL AND MANAGEMENT ENGINEERING

M.SC. IN ENGINEERING IN COMPUTER SCIENCE

**Neural Networks course**
**Final project**

A.Y. 2016/2017

# Sequence Classification

**Professors:**
Aurelio Uncini
Simone Scardapane

**Students:**
Giacomo Lanciano 1487019
Luca Marchetti 1475046
Leonardo Martini 1722989

# Contents

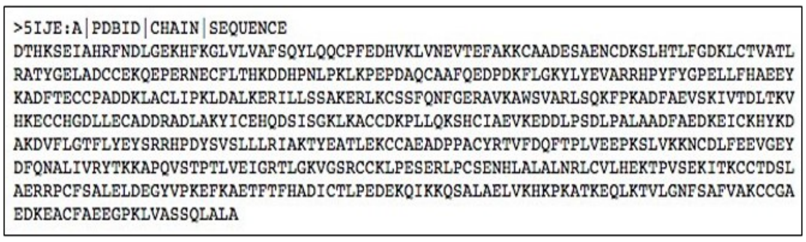
## 1 Introduction

In what follows, we present the development phases of the final project for *Neural Networks* course at Università degli Studi di Roma "La Sapienza", A.Y. 2016/2017.

In the world, a huge number of vegetable and animal proteins exists (in this paper we refer to 200.000 proteins). Each protein belongs to one and only one class. Each class is distinguished to the other by its characteristics.
Many approaches have been presented for the sequence classification problem (see [17] for a quick and to the point overview), including those based on pairwise similarity of sequences.
The generative methodology consists in building a model for a single protein family and then evaluating each candidate sequence to see how much it is compliant with the model, in the way that if the *fitness* is above a threshold, then the protein is classified as belonging to the family. Instead, discriminative approaches consider a different point of view: protein sequences are labelled examples (positive if they are in the family and negative otherwise) and a learning algorithm attempts to learn the distinction between the different classes. Notice that both positive and negative examples are possible using a discriminative approach, while generative approaches can only make use of positive training examples.
In this work, we follow the idea of using some discriminative approaches, in particular Support Vector Machines and Neural Networks, for protein classification. For the SVM, we use a *string* kernel (or *sequence-similarity* Kernel): the so-called **Spectrum kernel**, whose specifications and properties are depicted in [8]. The kernel is designed to be very simple and efficient to compute and does not depend on any generative model. Hence, we managed to produce an effective SVM-based classifier for protein sequences.
As an alternative approach, we build a protein classifier on top of a **Recurrent Neural Network**. It is a network made of neuron-like units, each one having a directed connection to every other unit. Each connection has a modifiable real-valued *weight*. Nodes can be partitioned in three different categories: *input* nodes, *output* nodes, and *hidden* nodes. For both approaches, an input sequence is represented through the corresponding list of *shingles* (see [16] for details).
The aim of this work is to present in details the underlying theory and to compare the performances of these different approaches. The source code we produced is available at our *Github* repository[4]. The data we collected are available at [5].

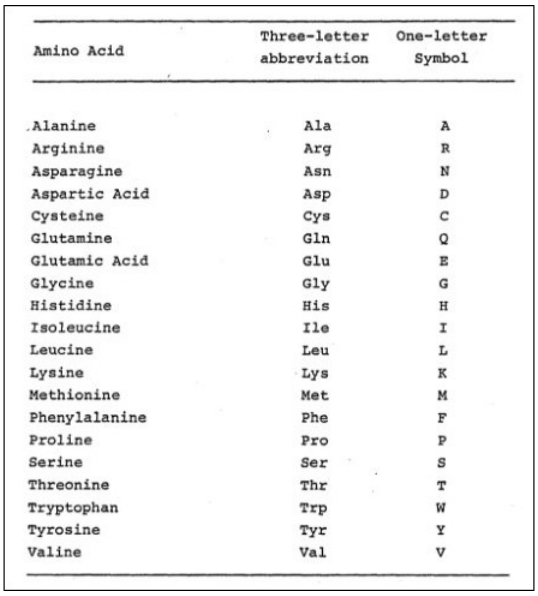
## 2 Data Retrieval

The first step of a classification problem is to build a dataset reporting the associations between the sequence (*predictor* variable) and the class label (*response* variable).
Every protein is uniquely identified by an ID called PDB, that is a string a four characters, that we assumed to be a primary key in our database. This information can be found in a single file at `http://www.rcsb.org/pdb/download/download.doFASTA`, together with the entire sequence of the protein. Each entry of the file is in FASTA format (see fig. 1), where the first line consists of the PDB identifier. Other information such as *chain* and *domain* are not relevant for our goal. The remaining lines represent the protein sequence. A sequence consists in a string of capital letters, each one corresponding to an amino-acid (the mapping is clearly shown in fig. 2).



```
>5IJE:A|PDBID|CHAIN|SEQUENCE
DTHKSEIAHRFNDLGEKHFKGLVLVAFSQYLQQCPFEDHVKLVNEVTEFAKKCAADESAENCDKSLHTLFGDKLCTVATL
RATYGELADCCEKQEPERNECFLTHKDDHPNLPKLKPEPDAQCAAFQEDPDKFLGKYLYEVARRHPYFYGPELLFHAEEY
KADFTECCPADDKLACLIPKLDALKERILLSSAKERLKCSSFQNFGERAVKAWSVARLSQKFPKADFAEVSKIVTDLTKV
HKECCHGDLLECADDRADLAKYICEHQDSISGKLKACCDKPLLQKSHCIAEVKEDDLPSDLPALAADFAEDKEICKHYKD
AKDVFLGTFLYEYSRRHPDYSVSLLLRIAKTYEATLEKCCAEADPPACYRTVFDQFTPLVEEPKSLVKKNCDLFEEVGEY
DFQNALIVRYTKKAPQVSTPTLVEIGRTLGKVGSRCCKLPESERLPCSENHLALALNRLCVLHEKTPVSEKITKCCTDSL
AERRPCFSALELDEGYVPKEFKAETFTFHADICTLPEDEKQIKKQSALAELVKHKPKATKEQLKTVLGNFSAFVAKCCGA
EDKEACFAEEGPKLVASSQLALA
```

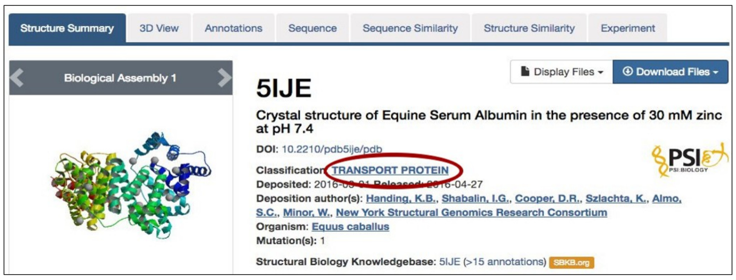
Figure 1: A protein sequence in FASTA format.



| Amino Acid | Three-letter abbreviation | One-letter Symbol |
|---|---|---|
| Alanine | Ala | A |
| Arginine | Arg | R |
| Asparagine | Asn | N |
| Aspartic Acid | Asp | D |
| Cysteine | Cys | C |
| Glutamine | Gln | Q |
| Glutamic Acid | Glu | E |
| Glycine | Gly | G |
| Histidine | His | H |
| Isoleucine | Ile | I |
| Leucine | Leu | L |
| Lysine | Lys | K |
| Methionine | Met | M |
| Phenylalanine | Phe | F |
| Proline | Pro | P |
| Serine | Ser | S |
| Threonine | Thr | T |
| Tryptophan | Trp | W |
| Tyrosine | Tyr | Y |
| Valine | Val | V |

Figure 2: A table showing the encoding for the amino-acids.

The problem is that this file does not contain the class the protein belongs to. In particular, we were interested in classifying proteins according to their *primary structure*. Therefore, given the PDB identifiers of the proteins, we had to integrate the database by collecting their labels using a Python crawler. For each protein, it issues a request to `http://www.rcsb.org/pdb/search/structidSearch.do?structureId=<PDB>` and performs the scraping of the resulting web page, extracting the classification label (as shown in fig. 3). The result of this integration process is the table shown in fig. 5a.



Structure Summary | 3D View | Annotations | Sequence | Sequence Similarity | Structure Similarity | Experiment

Biological Assembly 1

Display Files | Download Files

**5IJE**

**Crystal structure of Equine Serum Albumin in the presence of 30 mM zinc at pH 7.4**

**DOI:** 10.2210/pdb5ije/pdb

**Classification:** TRANSPORT PROTEIN

**Deposited:** 2016-0[illegible]-01 **Released:** 2016-04-27

**Deposition author(s):** Handing, K.B., Shabalin, I.G., Cooper, D.R., Szlachta, K., Almo, S.C., Minor, W., New York Structural Genomics Research Consortium

**Organism:** Equus caballus

**Mutation(s):** 1

**Structural Biology Knowledgebase:** 5IJE (>15 annotations) SBKB.org

Figure 3: The protein label to be scraped from the web page.

We also considered another kind of protein classification based on the *secondary structure*, that is is the three dimensional form of local segments of proteins. The two most common secondary structural elements are *alpha* helices and *beta* sheets, that are shown in the figure below. Secondary structure elements typically spontaneously form as an intermediate before the protein folds into its three dimensional tertiary structure.

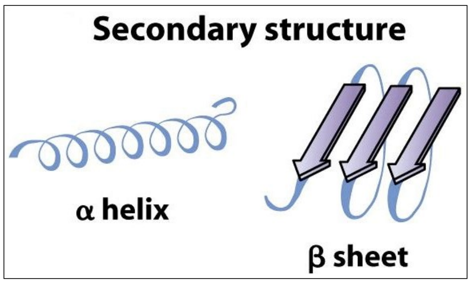
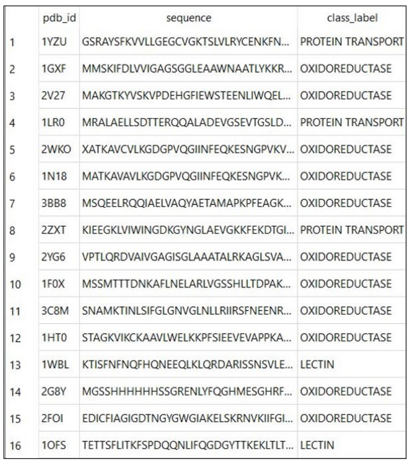


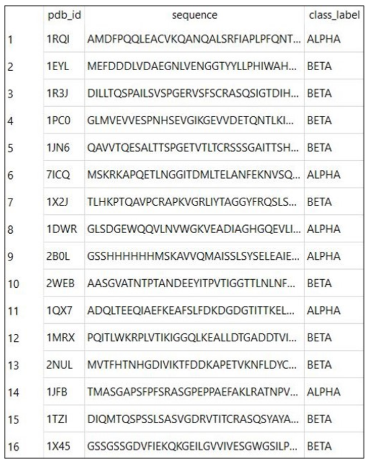
Figure 4: The most common proteins secondary structures.

For this purpose, we built another crawler to retrieve the new labels from SCOP (Structural Classification of Proteins) repository at `http://scop.mrc-lmb.cam.ac.uk/scop/data/scop.b.html`. The result of this merging process is the table shown in fig. 5b.



| | pdb_id | sequence | class_label |
|---|---|---|---|
| 1 | 1YZU | GSRAYSFKVVLLGEGCVGKTSLVLRYCENKFN... | PROTEIN TRANSPORT |
| 2 | 1GXF | MMSKIFDLVVIGAGSGGLEAAWNAATLYKKR... | OXIDOREDUCTASE |
| 3 | 2V27 | MAKGTKYVSKVPDEHGFIEWSTEENLIWQEL... | OXIDOREDUCTASE |
| 4 | 1LR0 | MRALAELLSDTTERQQALADEVGSEVTGSLD... | PROTEIN TRANSPORT |
| 5 | 2WKO | XATKAVCVLKGDGPVQGIINFEQKESNGPVKV... | OXIDOREDUCTASE |
| 6 | 1N18 | MATKAVAVLKGDGPVQGIINFEQKESNGPVK... | OXIDOREDUCTASE |
| 7 | 3BB8 | MSQEELRQQIAELVAQYAETAMAPKPFEAGK... | OXIDOREDUCTASE |
| 8 | 2ZXT | KIEEGKLVIWINGDKGYNGLAEVGKKFEKDTGI... | PROTEIN TRANSPORT |
| 9 | 2YG6 | VPTLQRDVAIVGAGISGLAAATALRKAGLSVA... | OXIDOREDUCTASE |
| 10 | 1F0X | MSSMTTTDNKAFLNELARLVGSSHLLTDPAK... | OXIDOREDUCTASE |
| 11 | 3C8M | SNAMKTINLSIFGLGNVGLNLLRIIRSFNEENR... | OXIDOREDUCTASE |
| 12 | 1HT0 | STAGKVIKCKAAVLWELKKPFSIEEVEVAPPKA... | OXIDOREDUCTASE |
| 13 | 1WBL | KTISFNFNQFHQNEEQLKLQRDARISSNSVLE... | LECTIN |
| 14 | 2G8Y | MGSSHHHHHHSSGRENLYFQGHMESGHRF... | OXIDOREDUCTASE |
| 15 | 2FOI | EDICFIAGIGDTNGYGWGIAKELSKRNVKIIFGI... | OXIDOREDUCTASE |
| 16 | 1OFS | TETTSFLITKFSPDQQNLIFQGDGYTTKEKLTLT... | LECTIN |

(a) *Sequences labelled according to primary structure.*



| | pdb_id | sequence | class_label |
|---|---|---|---|
| 1 | 1RQI | AMDFPQQLEACVKQANQALSRFIAPLPFQNT... | ALPHA |
| 2 | 1EYL | MEFDDDLVDAEGNLVENGGTYYLLPHIWAH... | BETA |
| 3 | 1R3J | DILLTQSPAILSVSPGERVSFSCRASQSIGTDIH... | BETA |
| 4 | 1PC0 | GLMVEVVESPNHSEVGIKGEVVDETQNTLKI... | BETA |
| 5 | 1JN6 | QAVVTQESALTTSPGETVTLTCRSSSGAITTSH... | BETA |
| 6 | 7ICQ | MSKRKAPQETLNGGITDMLTELANFEKNVSQ... | ALPHA |
| 7 | 1X2J | TLHKPTQAVPCRAPKVGRLIYTAGGYFRQSLS... | BETA |
| 8 | 1DWR | GLSDGEWQQVLNVWGKVEADIAGHGQEVLI... | ALPHA |
| 9 | 2B0L | GSSHHHHHHMSKAVVQMAISSLSYSELEAIE... | ALPHA |
| 10 | 2WEB | AASGVATNTPTANDEEYITPVTIGGTTLNLNF... | BETA |
| 11 | 1QX7 | ADQLTEEQIAEFKEAFSLFDKDGDGTITTKEL... | ALPHA |
| 12 | 1MRX | PQITLWKRPLVTIKIGGQLKEALLDTGADDTVI... | BETA |
| 13 | 2NUL | MVTFHTNHGDIVIKTFDDKAPETVKNFLDYC... | BETA |
| 14 | 1JFB | TMASGAPSFPFSRASGPEPPAEFAKLRATNPV... | ALPHA |
| 15 | 1TZI | DIQMTQSPSSLSASVGDRVTITCRASQSYAYA... | BETA |
| 16 | 1X45 | GSSGSSGDVFIEKQKGEILGVVIVESGWGSILP... | BETA |

(b) *Sequences labelled according to secondary structure.*

Figure 5: Resulting database tables.

## 3 Support Vector Machine

A Support Vector Machines (SVM) is a type of *supervised learning* algorithm. Given a set of labelled training vectors (positive and negative input examples), an SVM learns *decision boundaries* to discriminate between the classes. The result is a classification rule that can be used to classify new test examples. SVMs have exhibited excellent generalization performance (accuracy on test sets) in practice and have strong theoretical motivation in statistical learning theory.
Let, without loss of generality, $\mathcal{S}$ be the training set made of $m$ input vectors $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \{\pm 1\}$. A SVM consists in a linear classification rule $f$ such that given a pair $(\mathbf{w}, b)$, where $\mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$, we have

$$f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b$$

According to the rule, $\mathbf{x}$ is classified as *positive* if $f(\mathbf{x}) > 0$ and as *negative* if $f(\mathbf{x}) < 0$. Hence, the decision boundary corresponds to the *hyperplane* denoted by

$$\{\mathbf{x} \in \mathbb{R}^n : \langle \mathbf{w}, \mathbf{x} \rangle + b = 0\}$$

where $\mathbf{w}$ is a *normal* vector (with respect to the hyperplane) and $b$ is a bias. Requiring that $|\mathbf{w}| = 1$, we can define the *margin* of the SVM with respect to $\mathcal{S}$ as

$$m_{\mathcal{S}}(f) = \min_{\{\mathbf{x}_i \in \mathcal{S}\}} y_i f(\mathbf{x}_i)$$

The simplest kind of SVM is the *hard margin* classifier, which solves an optimization problem to find the linear rule $f$ with maximal margin. Thus, in the linearly separable case, the hard margin SVM finds the hyperplane that correctly separates the data and maximizes the distance to the nearest training points. Nevertheless, training sets are usually *non-linearly* separable, and the SVM optimization problem has to be modified to overcome this issue. Indeed, we need a trade-of between maximizing the margin and minimizing some classification error measure on the training set (i.e. a *soft* margin classifier).
One of the most interesting features of an SVM optimization problem is that it is equivalent to solving a *dual quadratic* programming problem. Indeed, in the *hard* margin case (without loss of generality), the decision boundary is found by solving for $\alpha_i$ the problem

$$\max \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \quad s.t. \quad \alpha_i \geq 0 \quad \forall\ i \in [1, m]$$

Parameters $\mathbf{w}$ and $b$ are then determined by the optimal $\alpha_i$ (and the training data).
This crucial equivalence lets us introduce the *kernel techniques* for the computation of the inner product $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$. In these settings, training data are labelled example $(x_i, y_i)$, where $x_i$ belongs to an input space $\mathcal{X}$ (e.g. a vector space, a strings space, etc.). Given that a feature map $\Phi : \mathcal{X} \to \mathbb{R}^N$ exists (where $\mathbb{R}^N$ is a possibly high-dimensional vector space called *feature space*), we get a kernel $K$ on $\mathcal{X} \times \mathcal{X}$ such that

$$K(x, y) = \langle \Phi(x), \Phi(y) \rangle$$

Hence, by replacing $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$ with $K(x_i, x_j)$ in the dual, we can use SVM in the feature space. This result leads to further computational improvements in the case we are able to *directly* compute the kernel without calculating the feature vectors.

### 3.1 Spectrum Kernel

To apply the SVM to protein classification problem, we use a particular kind of *string* kernel: the so-called **Spectrum Kernel**. As described in [8], this kernel function is designed to be very simple and efficient to compute.
In this case, the input space $\mathcal{X}$ is constituted by all finite length sequences of characters from an alphabet $\mathcal{A}$, such that $|\mathcal{A}| = l$. Given $k \geq 1$, the $k$-spectrum of a sequence of characters is the set of all the $k$-length (contiguous) sub-sequences that it contains.
The feature map $\Phi_k : \mathcal{X} \to \mathbb{R}^{l^k}$ is indexed by all possible sub-sequences $a$ of length $k$ from alphabet $\mathcal{A}$ and it is defined as

$$\Phi_k(x) = (\Phi_a(x))_{a \in \mathcal{A}^k}$$

where $\Phi_a(x)$ are the *occurrences* of sub-sequence $a$ in sequence $x$. Thus, $\Phi_k(x)$ is a weighted representation of the $k$-spectrum of input sequence $x$. Hence, the $k$-spectrum kernel is defined as

$$K_k(x, y) = \langle \Phi_k(x), \Phi_k(y) \rangle$$

Notice that feature vectors are *sparse*: the number of non-zero coordinates is bounded by $|x| - k + 1$. This fact leads to the possibility of applying very efficient approaches when computing the kernel function.
The best one is to build a *suffix-tree* for the collection of $k$-length sub-sequences of $x$ and $y$. Using a linear time construction algorithm for the suffix-tree k, we can build and annotate the suffix-tree in $\mathcal{O}(kn)$ time. Then, we can compute the kernel value by simply traversing the tree. Hence, the overall time complexity is in turn linear.
For the sake of simplicity, in our work we used an alternative version of the algorithm having a time complexity of $\mathcal{O}(n \cdot \log n)$ (see next section for additional details).

### 3.2 Implementation

In our implementation, we reused the SVM model provided by *scikit-learn* [11] Python API. The code for our SVM-based classifier can be found in the *machine_learning* directory of the attached project. It contains the following fundamental modules:

- *sequence_classifier_input.py*
- *kernel_functions.py*
- *model_performance_measures.py*

#### 3.2.1 Sequence classifier input

In this module we provide an utility class to build the training and test splits of the input dataset for the classifier.
The first step is to retrieve the entire labelled dataset from our relational database, selecting (at random) only entries referring to a predefined set of labels (for the sake of simplicity). For instance, we considered only protein belonging to *OXIDOREDUCTASE*, *LECTIN* and *PROTEIN TRANSPORT* classes. Then, for each protein sequence we compute the related shingles list (based on the $k$ parameter) and the *binary encoding* of each shingle, where each letter is translated in 26-bits form. In addition, since shingles lists could have different lengths, we perform a padding according to the maximum length. In the end, we partition the dataset in training and test splits using *scikit-learn* utilities.
Notice that the encoding is necessary for a correct usage of the *scikit-learn* SVM implementation, since string kernels are not supported.

### 3.2.2 Kernel functions

We propose two different implementations of the Spectrum Kernel algorithm. In both cases, starting from a training set (made of unordered shingles lists) of size $m$, we build a $m \times m$ matrix $K$ where $K_{ij}$ is the value of the spectrum kernel for the $i - th$ and the $j - th$ elements in the training set.
The first variant is naive and very inefficient, basically a one-to-one translation of the formal definition (thus, not used in model evaluation). Indeed, the computation of $K_{ij}$ is carried on by scanning the shingles of element $i$ and, for each of them, scanning the shingles of element $j$ counting how many shingles are equal (i.e. we use the same scheme of the algorithm for computing a *bag intersection*). In this case, the time complexity of computing $K_{ij}$ is $\mathcal{O}(n^2)$, where $n$ is the maximum number of shingles in a list.
For the second variant, we performed an optimization based on the idea of *merge algorithm* depicted in [7]. The computation of $K_{ij}$ is carried on by *simultaneously* scanning the shingles of element $i$ and the shingles of element $j$, previously sorted. Notice that in this way the two shingles lists are made of many sub-lists, each one containing all the occurrences of a unique shingle. At each step, we compare the values pointed by the two cursors:

- If values are equals, then we are scanning two sub-lists referring to the same shingle. Thus, both cursors are moved forward and both counters are increased by one until both sub-lists are consumed. Then, $K_{ij}$ is increased by the product of the counters.
- If values are not equals, then we move forward the cursor pointing to the minimum.
- If the end of a shingles list is reached, then we return $K_{ij}$.

It is trivial to see that the most costly step of this algorithm is the initial sorting. Hence, the time complexity of computing $K_{ij}$ is $\mathcal{O}(n \cdot \log n)$.
In our Python code, we made usage of a *Counter* object to get an occurrences dictionary of the elements in a shingles list (avoiding an explicit sorting and to actually scan the sub-lists). Since we had to perform a *k-fold cross validation* (within a *grid search* of the best parameters for the SVM), we improved this last variant by exploiting the fact that the Kernel matrix is symmetric in that case. Hence, in these settings, we can compute only the *upper triangular* part of the matrix.

### 3.2.3 Performance measures

After selecting the best parameters for the SVM model through a *k-fold cross validation* and a *grid search*, we plot a *confusion matrix* to evaluate the accuracy of the best model. By definition, a confusion matrix $C$ is such that $C_{ij}$ is equal to the number of observations known to be in group $i$ but predicted to be in group $j$.

# 4 Recurrent Neural Network

## 4.1 Beyond classical Neural Networks

Artificial neural networks (ANNs) are a computational model used in computer science and other research disciplines, which is based on a large collection of simple neural units (artificial neurons), loosely analogous to the observed behaviour of a biological brain's axons. Each neural unit is connected with many others, and links can enhance or inhibit the activation state of adjoining neural units. Each individual neural unit computes using summation function. There may be a threshold function or limiting function on each connection and on the unit itself, such that the signal must surpass the limit before propagating to other neurons. These systems are self-learning and trained, rather than explicitly programmed, and excel in areas where the solution or feature detection is difficult to express in a traditional computer program.

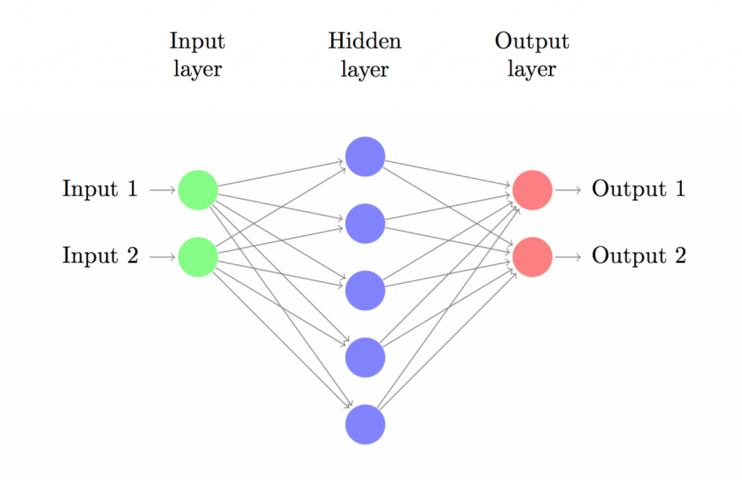


Figure 6: An example of 3-layer ANN.

In fig. 6 we can see an example of 3-layer neural network with one input layer, one hidden layer, and one output layer. The number of nodes in the input and output layers are determined respectively by the dimensionality of the data and the number of classes we have.
Conversely, we can arbitrarily choose the dimensionality of the hidden layer. The more nodes we put into the hidden layer the more complex functions we will be able to fit. Obviously, there is a trade-off between dimensionality and performances. Indeed, more computation is required to make predictions and learn the network parameters and a bigger number of parameters can lead to overfitting. However, the dimensionality of the hidden layer always depends on the specific problem.
We also need to pick an *activation function* for our hidden layer, that is the the one that transforms the inputs of the layer into its outputs. Common choices for activation functions are *tanh*, the *sigmoid* function, or *ReLUs*. In particular, *tanh* is one of the most used, since it behaves well in many scenarios. A nice property of these functions is that their derivative can be computed using the original function value. Since in our application the network has to output probabilities, the activation function for the output layer will be the *softmax*, which is a way to convert raw scores to probabilities.
The idea behind Recurrent Neural Networks (RNNs) is to make use of sequential information. In a classical neural network, the assumption is that all inputs (and outputs)

are independent from each other, but for many tasks this assumption leads to ineffective results. Indeed, humans do not start their thinking from scratch every second. For instance, if you want to predict the next word in a sentence, having knowledge about which words came before it can help to improve the prediction.

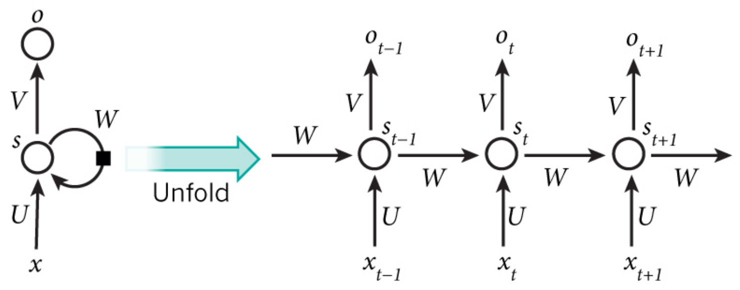


Figure 7: The structure of the computation in a RNN.

The *recurrent* behaviour of RNNs consists in performing the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have *memory* about what has been computed so far. In theory, they can make use of information in arbitrarily long sequences but in practice they are limited to looking back only a few steps. In fig. 7 you can see its *unfolded* computational structure, where:

- $x_t$ is the input at time step $t$.
- $s_t$ is the hidden state at time step $t$. It represents the *memory* of the network and it is computed on the basis of previous hidden state and the input at the current step: $s_t = f(Ux_t + Ws_{t-1})$. Notice that $f$ is the aforementioned activation function.
- $o_t$ is the output at time step $t$.

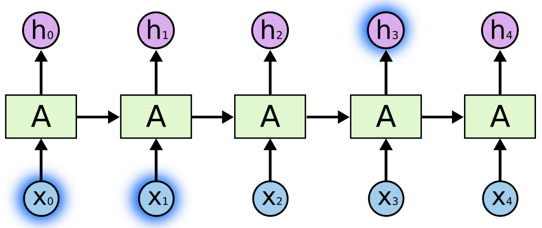
The big difference between RNN and classical neural networks, is that the former shares the same parameters ($U$, $V$, $W$) across all steps. This greatly reduces the total number of parameters we need to learn. However, *vanilla* RNN are not so good at capturing long-term dependencies, as it is required in many Sequence Classification problems (or NLP application in general). To overcome this issue, further improvements to the classical scheme have been proposed, such as *LSTM* and *GRU*.

## 4.2 LSTM

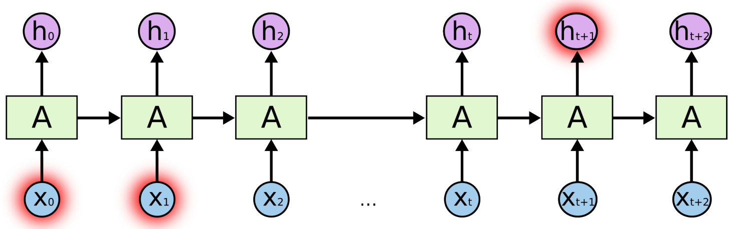
Long Short-Term Memory networks, usually called *LSTMs*, are a special kind of RNN that are able to handle long-term dependencies.

### 4.2.1 The problem of long-term dependencies

As said before, one of the core underlying ideas of RNNs is the capability to connect previous information to the present task. If RNNs could do that, they would be extremely useful. Sometimes, we only need to look at *recent* information to perform the present task. For instance, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in a small sentence where the gap between the relevant information and the place that is needed is small (see fig. 8a), simple RNNs can be effective.



(a) *Short-term context dependencies.*



(b) *Long-term context dependencies.*

Figure 8: RNN context dependencies (credits to [10]).

Conversely, there are also cases where we need more context (see fig. 8b). For instance, consider trying to predict the last word in a very long sentence where it is possible for the gap between the relevant information and the point where it is needed to become very large. In that cases, RNNs become unable to connect the information.

### 4.2.2 Long Short-Term Memory Networks

LSTMs are explicitly designed to avoid the long-term dependency problem. Indeed, the module of a LSTM cell is slightly different with respect to the one employed by classical RNNs (see fig. 9).

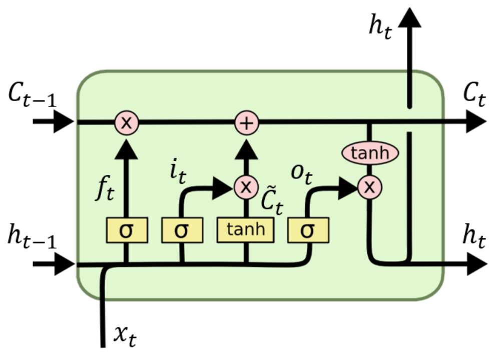


Figure 9: The structure of a LSTM cell (credits to [10]).

The key feature of LSTMs is the cell state, that is the horizontal line at the top of the diagram. It flows down the entire chain, with only some minor linear interactions with the other elements. LSTM is able to remove or add information to the cell state by means of the three *gates*, that consist in a *sigmoid* layer and a *point-wise multiplication* operation.
The first step in LSTM is to decide what information is going to be thrown away from the state. This decision is made by a sigmoid layer, the so-called *forget gate layer*. It

looks at $h_{t-1}$ and $x_t$, and outputs a number between 0 and 1 for each number in the cell state $C_{t-1}$. Intuitively, 1 stands for keeping all the information, while a 0 stands for dropping it all.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

The next step is to decide what new information is going to be stored in the cell state. This has two parts:

1. a sigmoid layer called the *input gate layer* decides which values will be update.
2. a *tanh* layer creates a vector of new candidate values, namely $\tilde{C}_t$, that could be added to the state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Then, the two are combined to create an update to the state. Indeed, the old state is multiplied by $f_t$, forgetting what has been decided to forget in previous steps. Then, $i_t * \tilde{C}_t$ is added. This is the new candidate values, scaled by how much we decided to update each state value.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finally, we need to decide what is going to be the output. It will be based on the cell state, but it will also be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we are going to output. Then, we put the cell state through *tanh* (to push the values to be between $-1$ and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$C_t = o_t * \tanh(C_t)$$

## 4.3 Implementation

In our implementation, we used the *TensorFlow* [2] framework to implement the RNN, reusing the existing implementation of the LSTM cell. The code for our RNN-based classifier can be found in the *neural_networks* directory of the attached project. It contains the following fundamental modules:

- *tf_glove.py*
- *tf_rnn.py*

### 4.3.1 TensorFlow overview

*TensorFlow* is an open source software library for numerical computation using data flow graphs. It has been developed at Google to meet their needs for systems capable of building and training neural networks to detect and decipher patterns and correlations, analogous to the learning and reasoning which humans use. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well. Indeed, it is currently used for both research and production at Google products. In May 2016 Google announced its tensor processing unit (TPU), a custom ASIC built specifically for machine learning and tailored for TensorFlow. The TPU is a programmable AI accelerator designed to provide high throughput of low-precision arithmetic (e.g., 8-bit), and oriented toward using or running models rather than training them. Google announced they had been running TPUs inside their data centers for more than a year, and have found them to deliver an order of magnitude better-optimized performance per watt for machine learning.

### 4.3.2 GloVe

Following an approach similar to what is described in [6], we used the *Global Vector* (*GloVe*) model to obtain a vector representation of the protein sequences in our training dataset.
As explained in [12], *GloVe* produces a representation of a *corpus* based on the co-occurrence statistics, setting the ground for an effective application of many data mining techniques that requires a vector space as input. Notice that in our case the corpus is composed by the *overlapping n-grams* of each sequence in the training dataset, looking at a sequence as a document and at its *n-grams* as the words contained in it. In this way, it is easier to effectively train the RNN to discover patterns among the sequences. Indeed, *GloVe* (such as other similar *word-embedding* techniques, like *word2vec* [9]) is a particularly effective approach in the context of the semantic and syntactic analysis of large text corpora. This model produces a vector space with meaningful substructure, and also outperforms related models on similarity tasks and named entity recognition. For what we are concerned in, this model is able to create vector representations (of fixed length) of the sequences in a way that the co-occurrence of two overlapping n-grams is taken into account. Thus, the RNN is able to discover this pattern and classify the protein on the basis of the corresponding vector. Here are the steps we went trough for building the dataset to be fed to the RNN:

1. train the *GloVe* model with the aforementioned sequences corpus.

2. for each sequence in the corpus, compute its overall embedding (i.e. vector representation) as the concatenation of the embeddings of its *n-grams* (that have a predefined length).

3. pack the resulting embeddings into a single matrix (that can be subsequently divided into training and test splits for the classifier).

Notice that the implementation of *GloVe* we employ in our code is again based on the *TensorFlow* framework.

### 4.3.3 Computation graph

After building the training input through the *GloVe* model, we are ready to create our *TensorFlow* (v1.0.1) implementation of the RNN. When working with a *TensorFlow* model we can identify two phases:

1. *building* phase, where the computation graph, consisting in all the calculations and functions that will be executed at runtime, is defined.

2. *execution* phase, where a *TensorFlow* session is created and the computation graph that was defined in the previous step is executed with the supplied data.

We start the first phase by defining two variables which will hold the training data and the training labels. These objects, so-called *placeholders*, are the fundamental building blocks in *TensorFlow* and their role is solely to serve as the target of feeds of data later on in the computation (i.e. they replace the tensors in the early stages). The dimensions of a placeholder for the training data are

$$[Batch\_Size,\ Sequence\_Length,\ Frame\_Dimension]$$

Conversely, a placeholder for the training labels (i.e. the desired results) has dimensions

$$[Batch\_Size,\ Labels\_Number]$$

Usually, the batch size (i.e. the number of input samples) is initially set at $None$ and determined at runtime.
Besides these two required placeholders, we added another one for the *dropout*. As described in [13], dropout is an effective technique for addressing the overfitting problem in large neural networks (with many parameters). The key idea is to randomly drop units (hidden or visible, along with their connections) from the neural network during training. This prevents units from co-adapting too much and provides a way of approximately combining exponentially many different neural network architectures efficiently. Indeed, model combination nearly always improves the performance of machine learning methods. With large neural networks, however, the obvious idea of averaging the outputs of many separately trained nets is expensive. Combining several models is most helpful when the individual models are architecturally different from each other.
The entire logic of our RNN is encapsulated into the *SequenceClassifier* class. It exposes an interface made of four fundamental functions:

- **prediction**: it builds the RNN starting from the basic LSTM cell. For each cell to be initialised, we need to supply a value for the hidden dimension (i.e. the neurons number). There is not a precise method for determining the best value: too high may lead to overfitting, very low may yield extremely poor results. Then, we call the function *dynamic_rnn()* that *unroll* the network, pass the data to it and return the output and the state. Notice that we are not interested in the latter, since every time we look at a new sequence it becomes irrelevant. After that, we transpose the output to switch batch size with sequence size, in order to retain the only last frame (i.e. we are only interested in the last output).
  The next step is to determine *weight* and *bias* parameters, and multiply the output with the weights and add the bias to it. In this way, we will have a matrix with a variety of different values for each class (for each element in the batch). What we are interested in is the probability score for each class (i.e. the chance that the sequence belongs to a particular class) computed through the subsequent *softmax* activation [15]. This function takes as input a vector $\mathbf{z}$ of $K$ real values and returns a probability distribution $\sigma(\mathbf{z})$ defined as

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \forall\ j \in [1, K]$$

- **cost**: it computes the loss of the model (i.e. its cost), that is the function that we aim to minimize and that determine how accurate is the prediction with respect to the actual results. In particular, we choose as cost function the *cross entropy* loss [14] defined as

  $$C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)]$$

  where $a$ is the output, $n$ is the total number of items of training data, the sum is over all training inputs $x$, and $y$ is the corresponding desired output. Notice that this function is *non-negative* and approaches to zero when the actual output is close to the desired output for all training inputs. Indeed, if for some input $x$ we have $y = 0$ and $a \approx 0$ (similarly for $y = 1$ and $a \approx 1$), then we have that the first term of the summation is none, while the second is $\ln(1-a) \approx 0$. Thus, it can be indeed interpreted as a cost function.
  In addition, cross-entropy loss avoids the so-called *learning slowing down* problem. In fact, the larger the error, the faster the neuron will learn, since the learning rate is controlled by the distance between the actual and the desired results.

- **optimize**: it defines the optimization strategy for the cost function to be minimized. In particular, we choose the implementation of the *Adam* optimizer (depicted in [3]) provided by *TensorFlow*, because of its computational efficiency and little memory requirements.

- **error**: it simply counts how many sequences in the test dataset are classified incorrectly.

The model is then completely defined and we can start the second phase by executing the computation graph to train the model. To do that, a *TensorFlow* session has to be started and the first step is to initialize all the previously defined variables.
We structured the training as a sequence of *epochs*, each one constituted by a fixed number of *steps*. At each step, a small sample of the training set (a so-called *batch*) is selected and fed to the model. Then, the error is computed with respect to the test set. At the end of each epoch, the mean error of its steps is computed. In this way, we are able to visualise the optimization process by plotting the (hopefully) decreasing value of the error.

## 5 Experiments

In what follows, we show the results of the experiments we conducted to validate the two sequence classification models we developed during this work: the *Spectrum Kernel* (plugged into a standard SVM) and the *Recurrent Neural Network*. Our aim is to highlight the differences in terms of *accuracy* and *performances* between the two solutions, considering various possible scenarios (i.e. tuning the problem settings and parameters as much as possible).
Notice that, due to the lack of powerful hardware and even tough we put a lot of effort in the *optimization* of our code, we had to restrict ourselves to a relatively small amount of data for the training phase. In particular for the training of the RNN, i.e. the real *bottleneck* of our workflow because of the huge amount of time required for each run (it may take even a whole day), we had to set up a virtual machine on *Google Cloud Platform*[1]. This cloud computing solution allowed us to run the RNN almost 24h per day and to (partially) overcome the memory issues we experienced on our personal machines. Thus, each test is based on datasets containing sequences from 2 different families of proteins, with a maximum of 1000 protein sequences per family.

### 5.1 Spectrum Kernel

For each experiment we run on the Spectrum Kernel, the SVM *hyper-parameters optimization* phase (performed through the *grid search* approach) gave use the same result (i.e. $C = 0.01$). Hence, the only parameter we can tune in this case is the length of the *n-grams* used to sketch the protein sequences (see *svm_main.py* script for details).
In fig. 10 you can see how the average accuracy changes according to different lengths of the *n-grams*. In fig. 11 the confusion matrices of each case are shown.

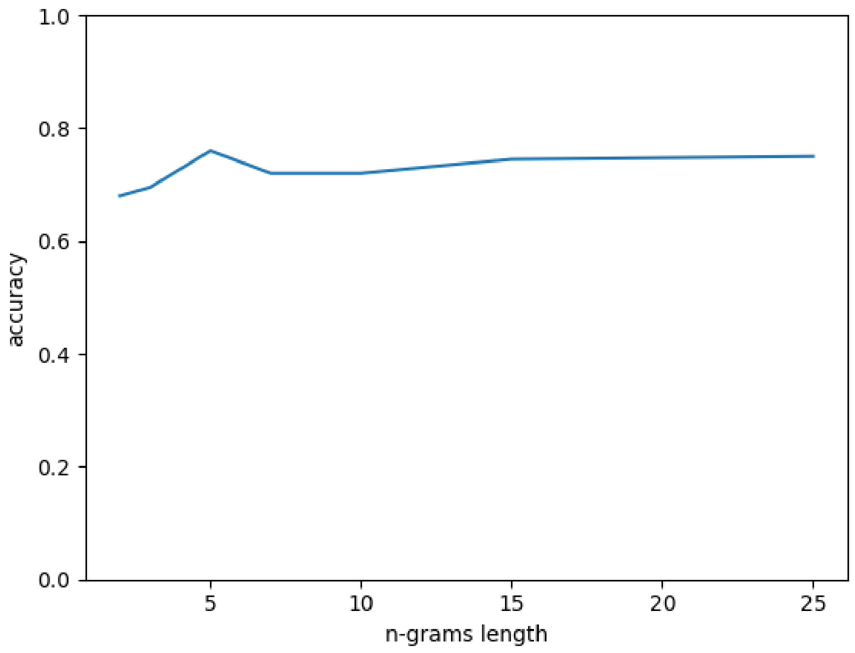
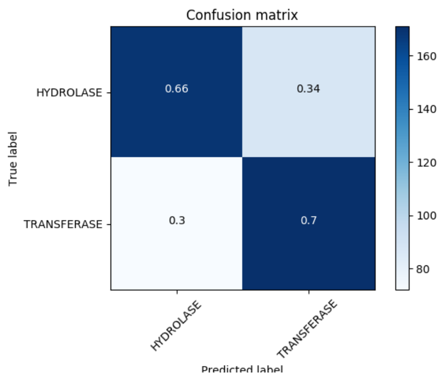
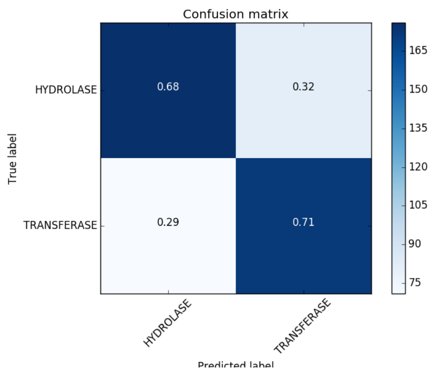


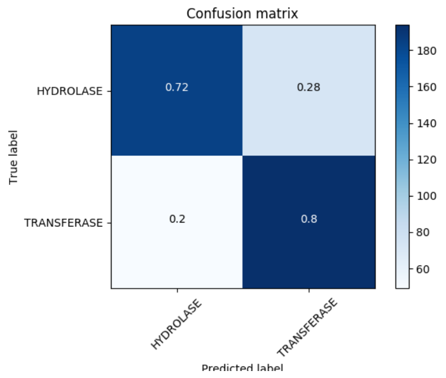
Figure 10: Average accuracy behaviour with respect to different n-grams lengths.



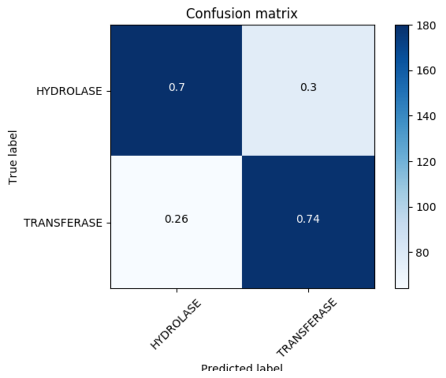
(a) *Confusion matrix for n-grams of length 2.*



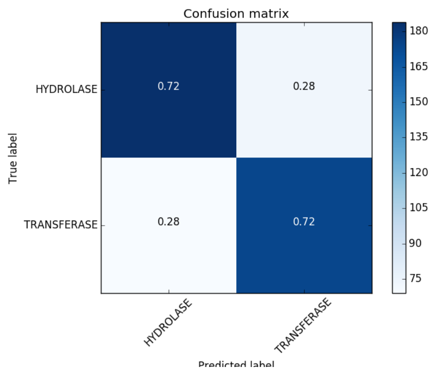
(b) *Confusion matrix for n-grams of length 3.*



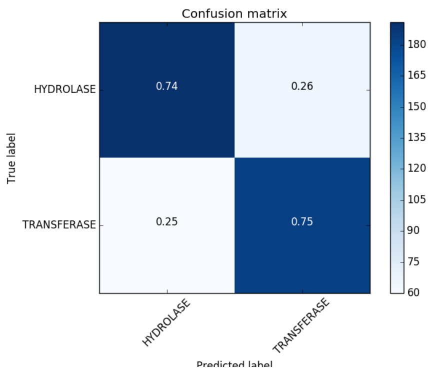
(c) *Confusion matrix for n-grams of length 5.*



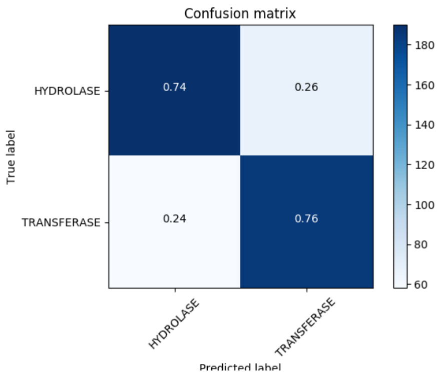
(d) *Confusion matrix for n-grams of length 7.*



(e) *Confusion matrix for n-grams of length 10.*



(f) *Confusion matrix for n-grams of length 15.*



(g) *Confusion matrix for n-grams of length 25.*

Figure 11: SVM confusion matrices.

As we can see, the average accuracy is not hugely affected by the variation of the n-grams length. Although, we can also notice a small pick around the value of 5 that could suggest: the more the number of n-grams representing the sequence (that is inversely proportional to n-grams length), the better the accuracy.

## 5.2 Recurrent Neural Network

The experiments we run on the Recurrent Neural Network are all structured in 10 *epochs* of 100 *time steps* each. At each time step, a small sample of the training dataset (a so-called *mini-batch*) is selected and used to perform the model optimization. Afterwards, the model error (i.e. the cost) is evaluated using the test dataset (see *tf_rnn.py* script for details).
Also in this case we monitored the accuracy of the model with respect to the variation of the n-grams size. In addition, we tested the network in two different *dropout* scenarios: *keep probability* set to 0.5 (i.e. dropping half of the connections) and set to 1 (i.e. drop no connections). The following plots show the results of the experiments conducted in these settings and in particular the behaviour of the **model error** during the training time steps. Notice that all of them refer to a number of neurons (per LSTM cells) equals to 100. Indeed, provided that the amount of training data is fixed to 2000 sequences (due to the aforementioned performance issues), it was not practical for us to further increase the number of neurons per cell: all our tests in such settings ended up in the Python process being killed by the OS due to **extreme memory usage**.

In figs. 12 to 14 we show the results for a dropout keep probability set to 0.5. We can see that the error decreases as the number of *n-grams* used to represent the sequences grows. Thus, in general, the shorter the n-grams, the better the accuracy of the model. Indeed, our experiments show that for n-grams of length 3 and 10 the (average per epoch) error decreases quite nicely during the training phase. Conversely, for n-grams of length 25 we experienced a significant error drop in the earlier epochs but afterwards it converges to an higher value.
In figs. 15 to 17, instead, we show the results of the experiments without dropout (i.e. keep probability set to 1). Recall that dropout is used to prevent the neural network from overfitting the training data. Our experiments show that in these cases the average error is higher and do not decrease smoothly as before (probably due to the overfitting). However, we can still notice a better behaviour for shorter n-grams.

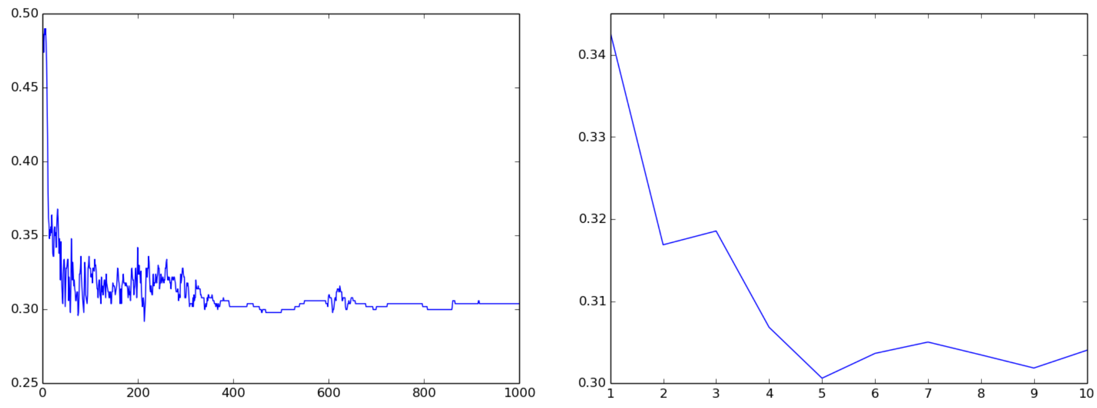
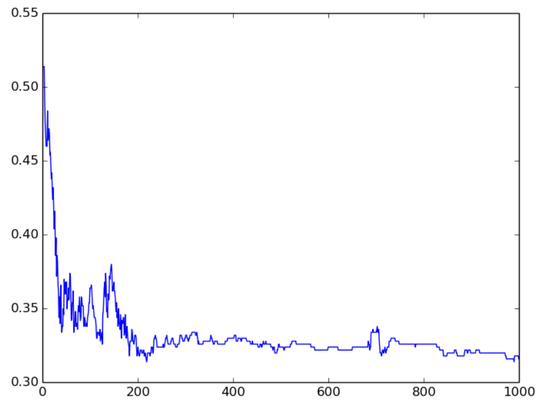
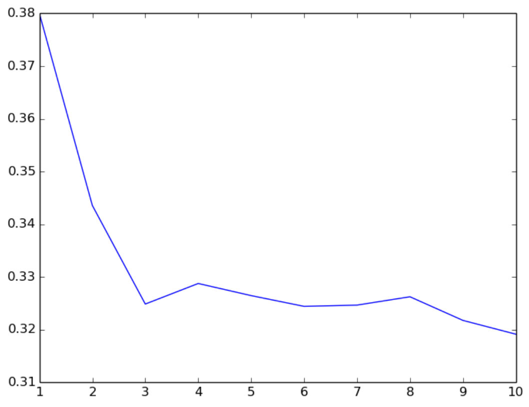

(a) *Model error per time step.*　　(b) *Average model error per epoch.*

Figure 12: Model error for n-grams of length 3 and dropout keep probability set to 0.5.

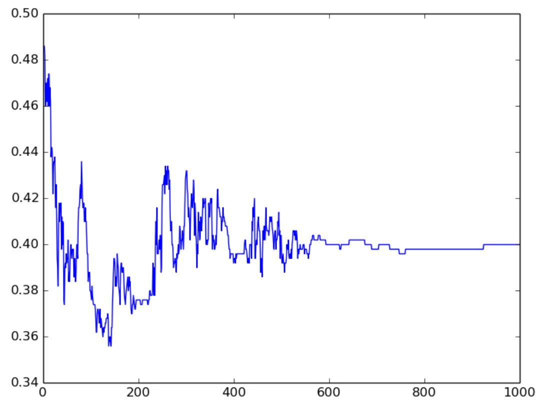


(a) *Model error per time step.*

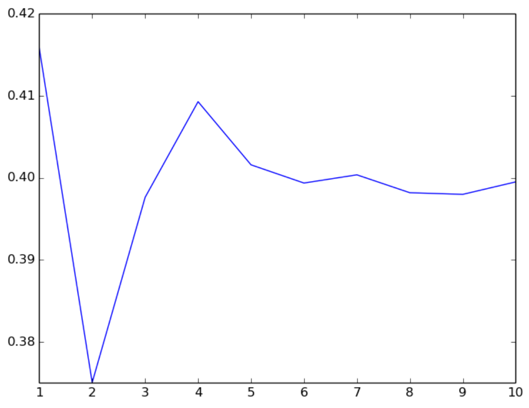


(b) *Average model error per epoch.*

Figure 13: Model error for n-grams of length 10 and dropout keep probability set to 0.5.

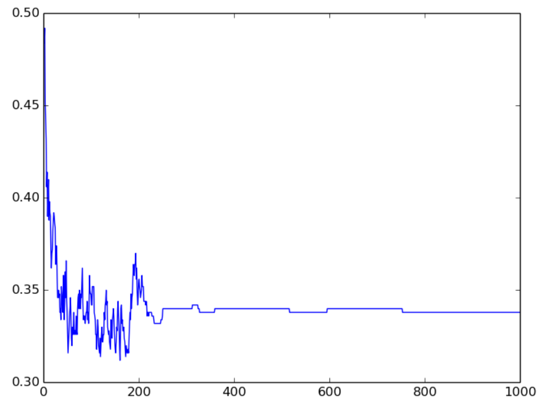


(a) *Model error per time step.*

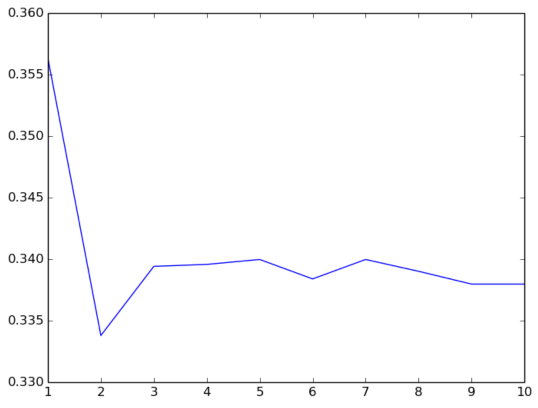


(b) *Average model error per epoch.*

Figure 14: Model error for n-grams of length 25 and dropout keep probability set to 0.5.

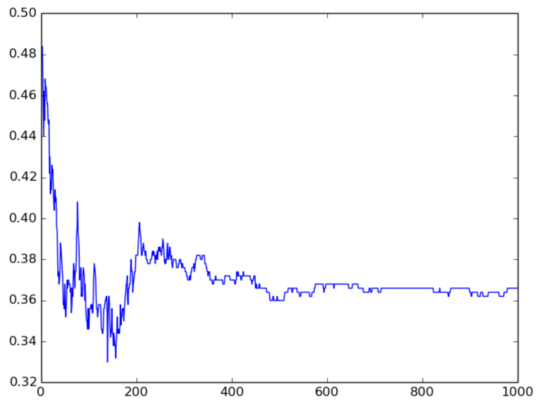


(a) *Model error per time step.*



(b) *Average model error per epoch.*

Figure 15: Model error for n-grams of length 3 and dropout keep probability set to 1.


(a) *Model error per time step.*

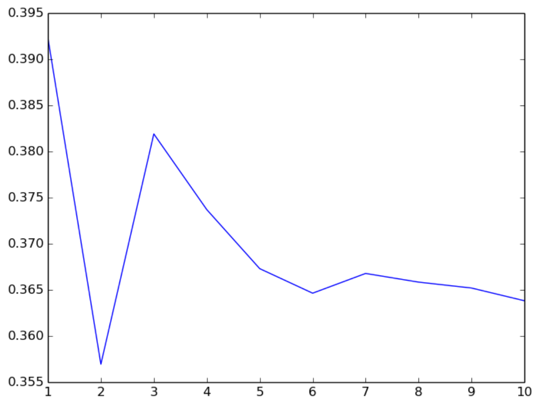
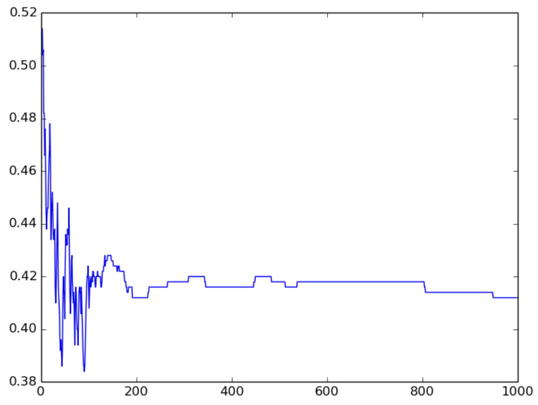

(b) *Average model error per epoch.*

Figure 16: Model error for n-grams of length 10 and dropout keep probability set to 1.


(a) *Model error per time step.*

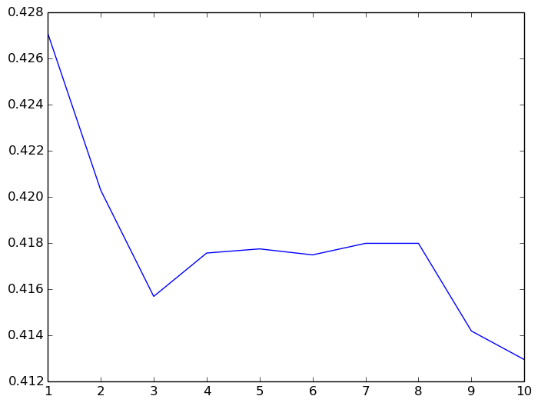

(b) *Average model error per epoch.*

Figure 17: Model error for n-grams of length 25 and dropout keep probability set to 1.

## 6 Conclusions and Future Works

The real bottleneck of our work has definitely been the lack of powerful and appropriate hardware to validate our implementations of the aforementioned models. Even though we highly optimized our code and set up a remote virtual machine on *GCP*[1] to run it 24/7 (to overcome the high training times), we could not test them on a massive amount of data.
Looking at our experiments, we can state that the SVM equipped with Spectrum Kernel performs slightly better (in average) in terms of accuracy and requires very little training time compared with the one required for the Recurrent Neural Network. Indeed, with the Spectrum Kernel we reached a maximum accuracy of 80% in a matter of minutes, while the RNN took even a whole day to complete the training and gave us at most a 70% of accuracy. However, what seems to be a constant, when moving from an approach to the other, is that increasing the size of the n-grams used to represent the sequences we get a degeneration of the accuracy. This result was expected since, intuitively, comparing two sequences divide in smaller chunks leads to an higher probability of discovering differences of recurrent patterns. Indeed, using both approaches, we got the best results when using the smallest n-grams (namely of size 5 for the Spectrum Kernel and of size 3 for the RNN).
In future works, our hope is to have the possibility to test our models on more powerful hardware, in order to better investigate the influence of their main parameters (e.g. the number of neurons of a single LSTM cell for the RNN) on the overall accuracy and to check the scalability of the approaches when dealing with bigger sizes of training datasets.

## References

[1] Google Inc. *Google Cloud Platform.* 2011. URL: `https://cloud.google.com/`.

[2] Google Inc. *TensorFlow.* 2015. URL: `https://www.tensorflow.org/`.

[3] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: `http://arxiv.org/abs/1412.6980`.

[4] Giacomo Lanciano, Luca Marchetti, and Leonardo Martini. *Sequence Classificationproject, code repository.* 2017. URL: `https://github.com/giacomolanciano/sequence-classification`.

[5] Giacomo Lanciano, Luca Marchetti, and Leonardo Martini. *Sequence Classificationproject, data repository.* 2017. URL: `https://drive.google.com/drive/folders/0B0FHkIijDk2hMVlUekJlUjVYM2M?usp=sharing`.

[6] Timothy K Lee and Tuan Nguyen. *Protein Family Classification with Neural Networks.* URL: `https://cs224d.stanford.edu/reports/LeeNguyen.pdf`.

[7] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets.* 2nd ed. Cambridge University Press, Dec. 2014. ISBN: 9781107077232.

[8] Christina S. Leslie, Eleazar Eskin, and William Stafford Noble. "The Spectrum Kernel: A String Kernel for SVM Protein Classification". In: *Proceedings of the 7th Pacific Symposium on Biocomputing, PSB 2002, Lihue, Hawaii, USA, January 3-7, 2002.* 2002, pp. 566–575. URL: `http://psb.stanford.edu/psb-online/proceedings/psb02/leslie.pdf`.

[9] Tomas Mikolov et al. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems.* 2013, pp. 3111–3119. URL: `https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf`.

[10] Christopher Olah. *Understanding LSTM Networks.* 2015. URL: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[12] Jeffrey Pennington, Richard Socher, and Christopher D Manning. "Glove: Global Vectors for Word Representation." In: *EMNLP.* Vol. 14. 2014, pp. 1532–1543. URL: `http://www.emnlp2014.org/papers/pdf/EMNLP2014162.pdf`.

[13] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435. URL: `http://dl.acm.org/citation.cfm?id=2627435.2670313`.

[14] Wikipedia. *Cross entropy — Wikipedia, The Free Encyclopedia.* [Online; accessed 27-April-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Cross_entropy&oldid=777199418`.

[15] Wikipedia. *Softmax function — Wikipedia, The Free Encyclopedia.* [Online; accessed 27-April-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Softmax_function&oldid=776879736`.

[16] Wikipedia. *W-shingling — Wikipedia, The Free Encyclopedia.* [Online; accessed 23-March-2017]. 2016. URL: `https://en.wikipedia.org/w/index.php?title=W-shingling&oldid=741428478`.

[17] Zhengzheng Xing, Jian Pei, and Eamonn Keogh. “A Brief Survey on Sequence Classification”. In: *SIGKDD Explor. Newsl.* 12.1 (Nov. 2010), pp. 40–48. ISSN: 1931-0145. DOI: 10.1145/1882471.1882478. URL: http://doi.acm.org/10.1145/1882471.1882478.