Network and Practical

Introduction to ML for Natural Language Processing

Daniele Passabì [221229], Data Science

Multi-class Classification of European Laws in Five Languages
How do Neural Networks and Statistical Models Behave?

Contents

1 Introduction

Natural Language Processing (NLP) originated in the 1950s as the intersection of artificial intelligence and linguistics [12]. It arose in response to the many challenges presented by natural language.

The ambiguity of human language makes it extremely difficult to write software that accurately determines the meaning of text or speech data. Creating hand-written algorithms and rules that can work on any text is very complex, given the presence of homonyms, homophones, sarcasm, unspoken content, idioms, metaphors, grammar and usage exceptions in virtually any text.

NLP combines computational linguistics with statistical models and machine learning in order to successfully process human language in the form of text or speech, understanding its meaning in ways that traditional algorithms are unable to do [10].

Natural Language Processing is used to solve a wide range of tasks: in this work we are interested in the one of multi-class text classification. Furthermore, our attention will be drawn to a particularly problematic aspect of the NLP field: the fact that most algorithms are designed and implemented for only 7 of the more than 7000 languages spoken in the world. These are English, Chinese, Urdu, Farsi, Arabic, French, and Spanish [18].

Our objective aims to test whether different languages can influence architectures such as neural networks or statistical models. To this end, we used a dataset containing 65000 European laws, officially translated into several languages, on which we were able to compare the behaviour of various architectures, both with regard to fine-tuning and their performance.

The project code is public and available on Github.

2 Data

The chosen dataset is called *MultiEURLEX* and is available on HuggingFace. It comprises 65000 European laws officially translated into 23 languages.

Of these available languages, five were selected, based on their adoption at European level:

- *English*, with 51% EU speakers
- *German*, with 32% EU speakers
- *Italian*, with 16% EU speakers
- *Polish*, with 9% EU speakers
- *Swedish*, with 3% EU speakers

For each text in the dataset, there are one or more annotated labels indicating the typology of the law. However, as can be understood by reading the paper on the dataset, these annotations are *granular* [8]. This allowed us to keep only the first level of labels, transforming the problem from a multi-label task to a multi-class one. This operation both simplified the classification and facilitated the evaluation of the architectures results.

This resulted in 21 different classes being initially present, shown in Figure 1 together with the number of occurrences (texts) available for each of them.

A preliminary analysis of the data, however, revealed that each law averaged around 1000 words, with slight fluctuations depending on the considered language. Being aware that this factor would be computationally burdensome during the training phase of the models, it was decided to choose 2000 random laws from each of the three classes with the most observations, i.e. class 2, 3 and 18.
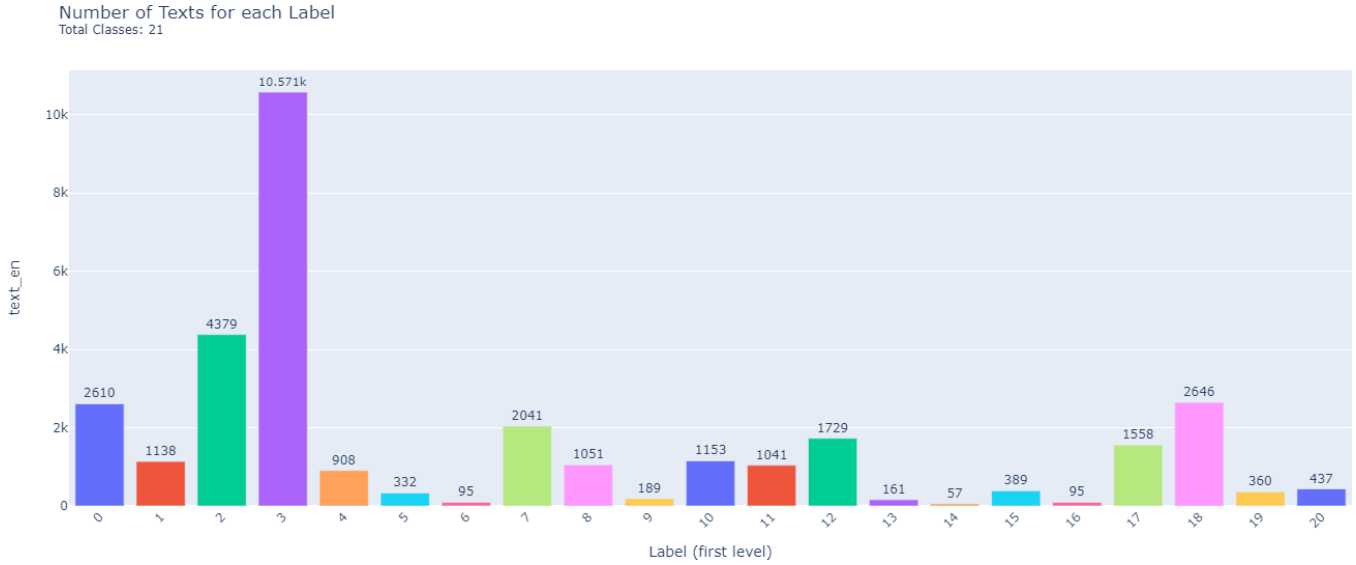
Figure 1: MultiEURLEX Dataset - Distribution of Classes and Texts Occurrences

Ultimately, the final dataset consists of 6000 laws, officially translated into 5 languages and belonging to 3 distinct classes.

3 Architectures

In order to test the impact of different languages, various architectures were chosen, belonging both to the world of neural networks and statistics.

For reasons of limited space, one architecture per typology is presented in detail: for neural networks, the functioning of an LSTM is explained in depth; for statistical models, the Linear Support Vector Classifier is illustrated. Nevertheless, the CNN architecture is briefly introduced.

3.1 Long Short Term Memory

The Long Short Term Memory (LSTM) is a kind of recurrent neural network (RNN). RNNs are a type of networks particularly suited to the use of sequential data, or time series.

A distinguishing characteristic from the traditional neural networks is their capacity to have a *memory*: they use information obtained from previous inputs to modify their behaviour on the current input and output. They keep the information of the former inputs in memory for an unspecified time, which depends on the weight of the network nodes and the specific data. Recurrent neural networks do not assume that input and output are independent, as traditional networks do, but make their output depend on the previous elements in the sequence. Figure 2 shows the typical structure of a feedforward neural network on the left, and that of a recurrent neural network on the right.
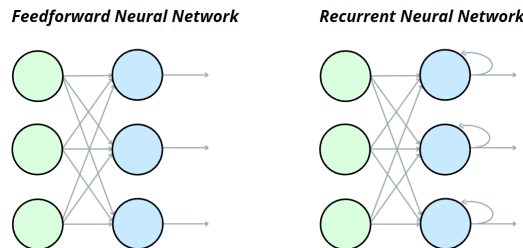

Figure 2: Structure of Feedforward and Recurrent Neural Networks

Recurrent neural networks use an algorithm of *backpropagation through time* to determine gradients. Backpropagation is at the basis of neural networks training, making possible the fine tuning of their weights, based on the results obtained in the previous epochs. The *gradient* refers to the derivative of a function that has more than one input variable; it measures the change in all the weights of the network with respect to the change error.

During this operation, recurrent neural networks tend to encounter two types of problems, both arising from the size of the gradient:

- *Vanishing Gradients*

  In some circumstances, the gradient is very small and continues to decrease until it becomes extremely close to 0. When this happens, the model stops learning, as its weights are no longer updated.

- *Exploding Gradients*

  Conversely, the gradient can become extremely large, altering the network weights unreasonably. This causes the model to become unstable and consequently underperforming.

Long Short Term Memory networks were conceived in response to the problems of vanishing and exploding gradient, proposing a possible solution to these issues [4]. A clear description of the inner workings of LSTMs can be found in the paper by Graves et al. [9] where it is detailed that:

> *"An LSTM layer consists of a set of recurrently connected blocks, known as memory blocks. These blocks can be thought of a differentiable version of the memory chips in a digital computer. Each one contains one or more recurrently connected memory cells and three multiplicative units - the input, output and forget gates - that provide continuous analogues of write, read and reset operations for the cells."*

In order, what happens is:

1. The cell input is multiplied by the activation of the input gate;

2. The model output is multiplied by the output gate activation;

3. Finally, the previous cell values are multiplied by the forget gate values.

The network is capable of interacting with its cells exclusively through the use of gates.

LSTMs achieve remarkably high performance on major and complex tasks; these include language modelling, speech recognition and machine translation [19].

3.2 Convolutional Neural Network

Convolutional neural networks (CNN) are one of the most renowned algorithms of deep learning. Under the deep learning category fall all those artificial neural networks that have more than one layer (and are thus called multi-layer architectures).

The layers of CNNs are usually the following: *Convolutional Layer*, *Non-Linearity Layer*, *Pooling Layer* and *Fully Connected Layer*. We will explain part of these layers in subsection 4.3, concerning the implementation of our architectures.

The models that use convolutional neural networks have shown over the years that they are able to achieve state of the art results in many Machine Learning problems. CNNs perform particularly well in image classification and computer vision tasks, but they have also proven powerful in problems of Natural Language Processing [1].

3.3 Linear Support Vector Classifier

In order to fully understand the idea behind the Linear Support Vector Classifier, it is first necessary to introduce some concepts that form its foundation.

First, let us introduce the notion of *hyperplane*. Intuitively, it is possible to think of the hyperplane as something that divides space into two sides, where it is straightforward to determine in which of the two sides a point is located. An example of hyperplane in two-dimensional space can be observed in Figure 3.

The visualisation also suggests how a hyperplane can be used to classify different observations. Let's suppose that the points in the figure represent the training set, i.e. the data initially available for training a model. It is then possible to construct a classifier in a very straightforward way: new points will be classified into *green* or *yellow observations* according to their position in space, given by X_1 and X_2.

Depending on how close the new observation is to the hyperplane it is also possible to have a certain degree of confidence in the correctness of the prediction. For points close to the separating hyperplane we are less certain about their class than for those further away.

There can exist an infinite number of different hyperplanes able to separate the green observations from the yellow ones. For this reason it is necessary to decide how to choose the *best* hyperplane. An appropriate choice is to use the *maximal margin hyperplane*, which is the hyperplane most distant from the training observations. It depends entirely on the so-called *support vectors*, i.e. the points closest to it that influence its position and orientation. See Figure 4 for an example.

Until now we have discussed a binary case in which the two classes were linearly separable. In the event that they are not, as it happens in many real world cases, it is no longer possible to find the maximum margin hyperplane. Even if it were feasible to find it, using it is not always the best solution: trying to perfectly separate the observations of the two classes makes the maximum margin hyperplane very sensitive to individual observations. It may be a better solution, in many cases, to consider a hyperplane that does not perfectly separate the training observations, but that allows to obtain a higher robustness regarding the individual observations and that correctly classifies most of the training observations.

Figure 3: Hyperplane in 2D case

Figure 4: Maximum Margin Hyperplane and Support Vectors (highlighted points)

This is indeed the objective of the Linear Support Vector Classifier. The model purposely allows the misclassification of certain observations, in order to correctly classify as many points as possible, while maximising the width of the margin.

Since in our specific case we are working with 3 classes, we are not in a binary classification scenario. We have therefore extended the binary approach to a multiclass approach using the scheme *One-VS-Rest*. It is a heuristic that allows us to split our multiclass dataset into multiple binary classification problems.
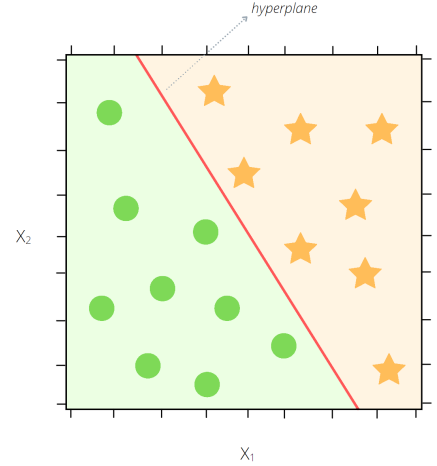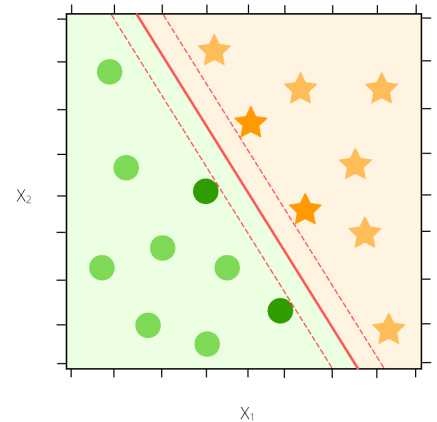
4 Experimental Setup

4.1 Data Cleaning and Preprocessing

The majority of data is prone to the presence of errors and noise. This is particularly true in the field of Natural Language Processing. Therefore, textual data must be subjected to pre-processing procedures before being given to the models.

The first critical step is data cleaning. In addition to the usual procedures such as removing duplicates, missing data, etc., when working with textual data, one often has to deal with texts that contain spelling mistakes, the use of jargon and symbols that are not useful for learning models. In our specific case, using a dataset of officially translated European laws, we do not have to worry about these issues.

Furthermore, texts always contain *stopwords*, commonly used terms that do not provide information, and which it is common practice to remove. However, these removal procedures use lists of words considered to be stopwords, but there are no universal rules. In order not to give any advantages to the most frequently used languages, such as English and German, we have decided not to remove them from the texts. Furthermore, working with recurrent neural networks like LSTMs, stopwords could increase the contextual sense of the networks and, contrary to what usually happens in other tasks, improve the performance of the models.

Another technique used when working with texts and natural language is to apply stemming or lemmatization procedures. These are described below.

- *Stemming*

  As can be found in the article *Development of a stemming algorithm* [11]:

  > "A stemming algorithm is a computational procedure which reduces all words with the same root (or, if prefixes are left untouched, the same stem) to a common form, usually by stripping each word of its derivational and inflectional suffixes."

- *Lemmatization*

  It is a procedure that removes inflectional endings and returns the base or dictionary form of a word.

  This technique ensures that similar words are reduced to the same word, which often leads to higher performance in models [2].

However, the same logic of stopwords applies to stemming and lemmatization procedures, which are therefore not applied to texts.

It is only fair to mention that the initial dataset has a very high quality: it was thus only necessary to lowercase all words, remove symbols and non-unicode characters, in order to ensure the proper functioning of future architectures.

4.2 Feature Extraction

Feature extraction describes the process of transforming raw data into numerical features that can be processed while retaining the information in the original dataset.

There are several ways of extracting information from data, and they change depending on the type of input that is being handled. In the context of this work, the primary focus is on the extraction of features from natural language. The procedures employed are outlined below.

Feature Extraction for Neural Networks

With regard to the creation of features for the neural networks used, namely long short term memory and convolutional neural network, we performed the following procedures:

- *Tokenization*

  It involves breaking up a raw text into smaller pieces. In our case, we break down each law text into words, the tokens.

- *Count Token Occurrences*

  We count the number of occurrences of each token (word) in our five corpuses.

  This is where we noticed the first difference between the languages: the number of different unique words used changes considerably, as can be seen in the Table 1. The German language, for example, uses almost three times as many words as English to write the same laws.

Language	Unique Words
English	31454
German	87084
Italian	44570
Polish	74691
Swedish	79973

Table 1: Total number of unique words in each language corpus

- *Removal of Uncommon Words*

  The number of different words in the texts is very high. This factor would certainly negatively affect the performance of the architectures or, at the very least, make training times extremely long.

  In order to avoid these possible drawbacks, all words that did not appear in the corpus at least 100 times were removed. Table 2 shows the number of words left for each language after this operation.

Language	Unique Words
English	3506
German	4216
Italian	4180
Polish	5255
Swedish	4010

Table 2: Total number of unique words in each language corpus that appears at least 100 times

  After this adjustment, the disparity between languages in the number of different words has softened, but is still present.

- *Encoding*

  In this last step, we created a mapping between vocabulary and index, in order to use it in the encoding of the texts. Here, we had to set a maximum token number for each text. Since the average number of words per law is about 1000, we used this value as an upper limit.

Feature Extraction for Linear Support Vector Classifier

Text Vectorization is a technique that consists of transforming textual data into numerical vectors that can be understood by machine learning architectures.

One of the most widely adopted techniques for text vectorization is *Term Frequency - Inverse Document Frequency* (TF-IDF), a statistical measure that assesses how relevant a word is to a document in a collection of documents.

In the case of laws classification, the text of each law represents one document.

The TF-IDF score for a word is obtained by multiplying the two metrics that compose the algorithm:

- *Term Frequency (TF)*

  In its simplest form, it refers to the number of times a word appears in a document.

- *Inverse Document Frequency*

  It refers to how common a word is in the entire set of sentences. The more frequent a word is, the closer its value is to 0.

  It is possible to calculate the IDF of a word by obtaining the total number of documents, dividing it by the number of documents containing that given word and computing the logarithm.

Multiplying these two values makes it possible to obtain the TF-IDF of a word in a document. The higher the value, the greater the relevance of the word within a document.

The TF-IDF is an efficient solution to the problem of *stopwords*: since they are frequent words in all documents, they have a value very close to zero and do not become influential for the models. The intriguing aspect is that this logic should be applicable to any language.

If a word is repeated many times within the same document, on the contrary, it might indicate its importance. The algorithm is able to detect these subtleties and is a particularly effective way of preparing the input for future architectures.

4.3 Implementing the Architectures

A brief description of the implementation of the architectures presented in Section 3 follows. In the case of neural networks, we focus on the employed layers.

Long Short Term Memory

The architecture was implemented using the `PyTorch` framework. In the development of the network, the following layers were used:

- *Embedding Layer*

  This layer is often used to store and retrieve word embeddings using indexes. The input of the module is a list of indexes and the output is the corresponding word embedding [15].

- *LSTM Layer*

  As one can easily deduce from its name, this module is the heart of the network. It applies a multi-layer long short-term memory to an input sequence [17].

- *Linear Layer*

  Its purpose is to apply a linear transformation to the input data [16].

- *Dropout Layer*

  During the training phase, some elements of the input tensor are zeroed with probability p, using samples of a Bernoulli distribution.

  This technique has proven to be effective in regularising and preventing co-adaptation of neurons [14], which refers to those cases in which some neurons are highly dependent on others.

Convolutional Neural Network

This architecture, like the illustrated one above, has also been implemented using the `PyTorch` framework. Note: some layers are the same as those employed for the LSTM and will thus not be explained again.

- *Embedding Layer*
- *Convolutional 1D Layer*

  Convolutional layers are one of the main building blocks used in convolutional neural networks. The term *convolution* refers to the application of a filter to an input, which then produces an activation. A repeated application of the same filter to an input produces an activation map, also called a feature map, which indicates the position and strength of a detected feature in an input.

  It is a linear operation involving the multiplication of a set of weights with the input, a common operation in neural networks [6].

- *Max Pooling 1D Layer*

  Pooling layers make it easy to down-sample feature maps. They summarise the presence of features in patches of the feature map.

  One of the most widely used pooling levels is max pooling, partly due to its high performance. In this instance, the pooling operation calculates the maximum value in each patch of each feature map [5].

- *Linear Layer*
- *Dropout Layer*

Linear Support Vector Classifier

This architecture was developed using the `scikit-learn` framework. Since it is a more high-level library than `Pytorch`, the implementation was simpler and more straightforward. Most of the work consisted in obtaining suitable features for the model, which was performed via the TF-IDF algorithm presented in subsection 4.2.

4.4 Training Regime

During the course of the project, particular emphasis was placed on fine-tuning the models. A detailed explanation of what was performed for each architecture follows.

Dataset Splits

Prior to proceeding with the model training phase, each dataset was divided upstream into three sets:

- *training set* (3840 texts)
- *validation set* (960 texts)
- *test set* (1200 texts)

Saving the Best Model

Neural Networks During each epoch the models were trained on the training set and validated on the validation one. As loss we have opted for *cross entropy*, which is considered suitable for cases of multiclass classification, both when working with statistical models and NNs [3].

We implemented an automatic mechanism capable of saving the model weights of the best epoch based on the average accuracy of the three classes, along with a dataframe containing the training results. The latter was employed to create data visualizations and better understand what occurred during the training phase, thus making it possibile to observe differences between languages. We will further discuss training results in Section 5.

Statistical Models Saving the results of the linear support vector classifiers was straightforward, as we did not have to handle the different epochs. The *squared hinge loss*, proposed as default by the `scikit-learn` framework, was chosen.

Long Short Term Memory

For each of the five available languages, 16 models were trained, i.e. the combination of a grid search performed on the hyperparameters shown in Table 3.

Hyperparameter	Employed By	Values Tested
Embedding Dim	Embedding Layer	1024, 2048
Hidden Dim	LSTM Layer	1024, 2048
Dropout Probability	Dropout Layer	0, 0.1
Learning Rate	Adam Optimizer	0.0001, 0.001

Table 3: Hyperparameters tested for LSTM networks

Each model was trained for a total of 50 epochs, using a batch size of 64. As a note, we are aware that in order to obtain more robust and reliable results it would have been ideal to increase the number of hyper-parameter values tested, along with the number of epochs. Training times, however, would have risen exponentially. Each complete search grid, which trains 16 models over 50 epochs, takes between 14 and 16 hours on an *RTX2070 Super* GPU, slightly varying according to the language of the texts.

Convolutional Neural Network

Convolutional neural networks proved to be less computationally demanding than LSTMs. This allowed us to test a higher number of hyperparameter values, with a total of 96 models per language (Table 4).

Hyperparameter	Employed By	Values Tested
Embedding Dim	Embedding Layer	1024, 2048
Kernel Size	CNN Layer, Max Pooling Layer	3, 5, 7
Stride	CNN Layer, Max Pooling Layer	1, 3
Padding	CNN Layer	1, 3
Dropout Probability	Dropout Layer	0, 0.1
Learning Rate	Adam Optimizer	0.0001, 0.001

Table 4: Hyperparameters tested for Convolutional Neural Networks

As with the LSTMs, the CNNs were also trained for 50 epochs, using a batch size of 64.

Linear Support Vector Classifier

For this architecture, there are no performance concerns. We therefore trained 7 different models for each language, each with a different value of C, the model's only hyperparameter.

C is the parameter responsible for *regularization*, which is inversely proportional to its value. Thus, for high values of C the optimiser will choose a hyperplane with a narrower margin if it correctly classifies all points in the training set. Conversely, a small value of C will cause the optimiser to look for a separating hyperplane with a larger margin, even if this hyperplane incorrectly classifies a greater number of points.

The values that we tested for the C hyper-parameter are the following: 0.1, 1, 5, 10, 50, 100, 1000.

5 Results

5.1 Training Times

Throughout this project we have always kept track of the training times of the models. We believe that this can provide useful insights regarding the differences that are present due to the languages, as well as those present due to the different architectures. A table of training times is shown in Figure 5.

[LSTM] Training Times			
Language	*Fine-Tuning Time*	*Mean Time per Model*	*Mean Time per Epoch*
English	15h 44m	0h 59m	1m 11s
German	16h 36m	1h 02m	1m 14s
Italian	14h 20m	0h 54m	1m 05s
Polish	15h 38m	0h 59m	1m 10s
Swedish	16h 12m	1h 01m	1m 13s

[CNN] Training Times			
Language	*Fine-Tuning Time*	*Mean Time per Model*	*Mean Time per Epoch*
English	2h 06m	1m 19s	1,58s
German	2h 12m	1m 23s	1,66s
Italian	2h 06m	1m 19s	1,58s
Polish	2h 10m	1m 21s	1,62s
Swedish	1h 58m	1m 14s	1,48s

[LSVC] Training Times		
Language	*Fine-Tuning Time*	*Mean Time per Model*
All	2m 43s	5s

Figure 5: Training Times of the Architectures

As can be observed from the illustration, the differences in time due to architectures are significantly greater than those due to the language used. There are, however, considerable time fluctuations in CNNs when training a model with German or Swedish texts, compared to other languages.

It is worth noting, however, that although we have attempted to train the models consistently under the same conditions, the computer may autonomously have launched background processes beyond our control that affected training times.

A small technical note: the two neural networks were trained on an *RTX2070 Super* GPU, while the linear support vector classifiers on an *i7-9700K CPU*.

5.2 Training Results

Before discussing the performances of the models, it is of interest to focus on the training phase of the architectures. Figure 6 shows the best hyperparameters chosen for each architecture of each language, together with the best epoch (in the case of neural networks) and the validation accuracy, obtained as the average of the precision of the three different classes.

It is immediately evident the disparity in the number of hyperparameters present in the neural networks compared to the statistical model, which only has C. The higher the number of hyper-parameters, the more difficult it is to conduct an exhaustive fine-tuning, capable of really giving us the best performance the model can offer.

It is interesting to note that the same values are always chosen for certain hyper-parameters, as in the case of *embedding dimension* and *dropout p* for LSTMs, *stride* for CNNs and *learning rate* for both. Of the seven values of C tested, 1 always proved to be the best for the LSVCs.

Another aspect we noticed from the empirical results was that even when changing the value of C by several orders of magnitude, the performance of the model varied negatively by a maximum of 2%. If we consider that, on the other hand, for "unlucky" combinations of hyperparameters the performances of the neural networks halves it is possible to realise how robust the linear support vector classifier is.

[LSTM] Best Hyperparameters, Best Epoch and Val Accuracy								
Language	Batch Size	Vocab Size	Emb. Dim	Hidden Dim	Learn. Rate	Dropout	Best Epoch	Val Accuracy (mean)
English	64	3506	1024	2048	0,001	0,0	31	0,8488
German	64	4216	1024	1024	0,001	0,0	42	0,7774
Italian	64	4180	1024	2048	0,001	0,0	34	0,8538
Polish	64	5255	1024	1024	0,001	0,0	13	0,8462
Swedish	64	4010	1024	1024	0,001	0,0	48	0,8459

[CNN] Best Hyperparameters, Best Epoch and Val Accuracy										
Language	Batch Size	Vocab Size	Emb. Dim	Kernel Size	Stride	Padding	Learn. Rate	Dropout	Best Epoch	Val Accuracy (mean)
English	64	3506	1024	5	1	3	0,001	0,1	46	0,8874
German	64	4216	2048	5	1	1	0,001	0,1	31	0,8731
Italian	64	4180	2048	7	1	1	0,001	0,0	20	0,8893
Polish	64	5255	2048	5	1	3	0,001	0,0	12	0,8929
Swedish	64	4010	1024	7	1	1	0,001	0,1	15	0,8698

[LSVC] Best C and Val Accuracy		
Language	C	Val Accuracy (mean)
English	1,0	0,9112
German	1,0	0,9121
Italian	1,0	0,9096
Polish	1,0	0,9123
Swedish	1,0	0,9090

Figure 6: Best Hyperparameters, Best Epoch (when available) and Validation Accuracy for each Architecture

Further interesting insights can be obtained by exploring the two visualisations made for the training phases of the best LSTMs and CNNs, which can be found in Figure 8 and 9. The first column shows the graphs depicting training and validation loss for each language. The second shows the accuracy of each class as the epochs increase. In both views, the vertical green dashed line indicates the best epoch chosen for that specific model.

In the case of the LSTMs, one can see how the *training loss* falls, more or less quickly and steadily depending on the language, until it almost reaches 0, a state that is usually associated with overfitting.

The *validation loss*, on the other hand, tends to fall and then rise again, assuming its usual elbow shape. It can be seen that in most cases the mean accuracy of the three classes is the highest just before the validation loss starts to rise again.

In the case of CNNs, however, something a little unusual happens. Both the training loss and the validation loss decrease and remain more or less constant in a value between 0.6 and 0.8. This could happen since we only used one convolutional layer in our architecture, which may not be able to fully capture the complexity of the data.

To confirm this thesis, it can be seen in the second column how the performance on the three classes is not very stable and continuously fluctuates. In the LSTMs this does not happen: after a certain number of epochs, the performance of the three classes stabilises (excluding the case of German).

5.3   Model Performances

Figure 7 shows the results obtained from the best architectures for each language on the test set. Colours are used in the visualisation to highlight the differences in the models performances.

[LSTM] Test Set Results

Language	Loss (global)	Accuracy (global)	Accuracy (mean)	Accuracy [0]	Accuracy [1]	Accuracy [2]
English	0,5561	0,8217	0,8213	0,8307	0,8474	0,7859
German	0,9223	0,7575	0,7601	0,7025	0,7632	0,8146
Italian	0,5415	0,8133	0,8138	0,8032	0,8263	0,8120
Polish	0,6351	0,8075	0,8086	0,7849	0,8237	0,8172
Swedish	0,9211	0,8192	0,8196	0,8124	0,8500	0,7900

[CNN] Test Set Results

Language	Loss (global)	Accuracy (global)	Accuracy (mean)	Accuracy [0]	Accuracy [1]	Accuracy [2]
English	0,7085	0,8425	0,8415	0,8627	0,8237	0,8381
German	0,7286	0,8208	0,8211	0,8146	0,8368	0,8120
Italian	0,6919	0,8525	0,8517	0,8696	0,8342	0,8512
Polish	0,7137	0,8375	0,8381	0,8261	0,8737	0,8146
Swedish	0,7139	0,8308	0,8291	0,8696	0,8395	0,7781

[LSVC] Test Set Results

Language	Loss (global)	Accuracy (global)	Accuracy (mean)	Accuracy [0]	Accuracy [1]	Accuracy [2]
English	-	0,8792	0,8783	0,9078	0,8601	0,8670
German	-	0,8758	0,8750	0,9117	0,8594	0,8539
Italian	-	0,8775	0,8768	0,9074	0,8528	0,8701
Polish	-	0,8800	0,8792	0,9141	0,8668	0,8568
Swedish	-	0,8850	0,8842	0,9125	0,8590	0,8811

Figure 7: Training Times of the Architectures

LSTM LSTM networks have the worst performance despite the highest training times.

It is interesting to observe how they have the most difficulties classifying German and Swedish texts, which are characterised by a higher loss compared to the other languages and, in the case of German, significantly lower performance. We would certainly need the opinion of an expert linguist, but our assumption is that German has a high number of compound words, which for people immediately acquire a meaning, but for

models are completely different from the two words that compose them. Moreover, in some cases the order of the sentence syntagmas changes completely from the usual (example: the verb is placed at the end of the sentence).

It can be observed from the results that, with the exception of one language, class 1 is always the one with slightly higher accuracy.

CNN Similarly to what happened in the LSTMs, also in the CNNs the lowest performances are achieved by the German and Swedish models, although the difference between them and the performance of the other models is milder. With this type of network, however, the loss values are very close, with no substantial difference dependent on language.

LSVC It is very interesting to observe how a statistical model that trains in five seconds on any CPU can compete and outperform neural networks that require hours of training on a dedicated GPU. According to the literature, this can happen thanks to TF-IDF, which is very reliable and leads to higher results than the words embedding employed by neural networks [7].

The LSVC models do not seem to be influenced by language, and obtain very satisfactory results in all five cases, with little to no difference between them.

It is worth noting that, in contrast to LSTMs, this model predicts class 0 texts with greater ease and consequent accuracy.

6 Discussion

6.1 About the Results

During the course of the project, we trained different architectures, belonging to related but different worlds, that of statistics and machine learning.

Our purpose was to understand whether and to what extent texts in different languages would change the training times and performances of the models. We were able to demonstrate the robustness of TF-IDF, which allowed the Linear Support Vector Classifiers to perform very well regardless of the language used.

The analysis also gave us an opportunity to reflect on the fragility of neural networks, which require to be fine-tuned on a high number of parameters and for considerable time in order to have satisfactory results. LSTMs more than CNNs seem to be susceptible to the language they are trained on, presumably since they are sequential models and as such are more influenced by how the sentence is formed, rather than just the words contained within it.

6.2 Ideas for Expanding the Project

During the course of the project, choices were made to simplify the task and the training time required by the models. This was mainly done for reasons of limited time and means at our disposal: having only one graphics card available, certain choices proved necessary.

The following is a list of suggestions and ideas through which the project could be expanded, if we had the means to do so.

- It would first be interesting to use all the texts of the laws at our disposal and repeat the experiment.
- Furthermore, it would also be worthwhile to consider the multi-label task and observe how the architectures behave in different languages.

- Having multiple GPUs available, the experiment could be repeated using all 23 languages from the initial dataset.

- A further idea is to try applying stopwords removal, lemmatization and/or stemming procedures to texts. The aim here is to see if and how the performance of the models changes and if the procedures favour the most common languages.

- With regard to the fine-tuning phase, as already mentioned in the relevant subsection 4.4, it would be advisable to extend the range of values used for each hyper-parameter in order to obtain more robust results.

In this regard, there are some implementable refinements that could save time during the fine-tuning phase:

- An *early stopping* mechanism could be implemented, which interrupts network training if after n epochs the validation loss has not decreased;

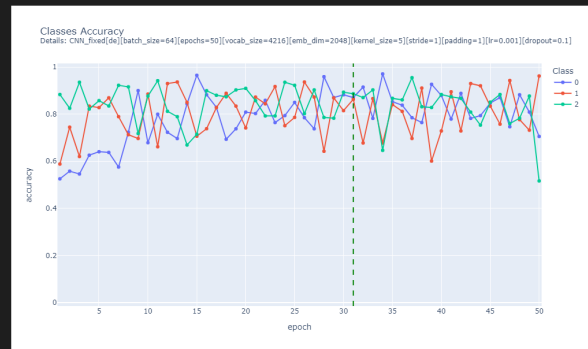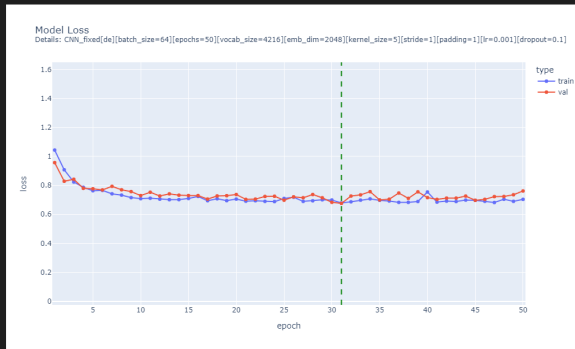- One could use more efficient tuning algorithms than the "brute" grid search, such as the one proposed by the `Keras` framework [13].

LSTM Training Results

Models Loss
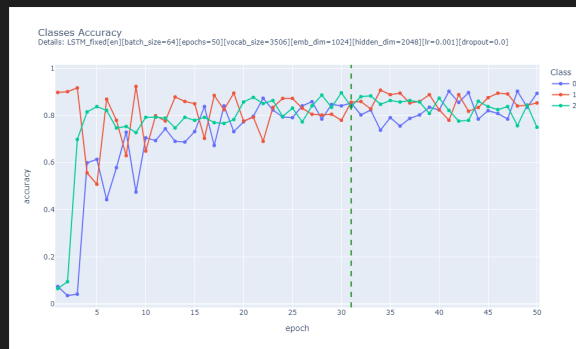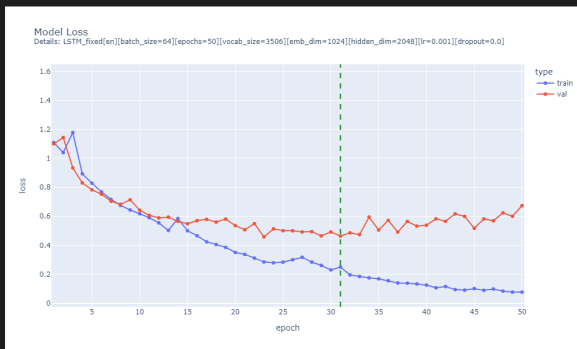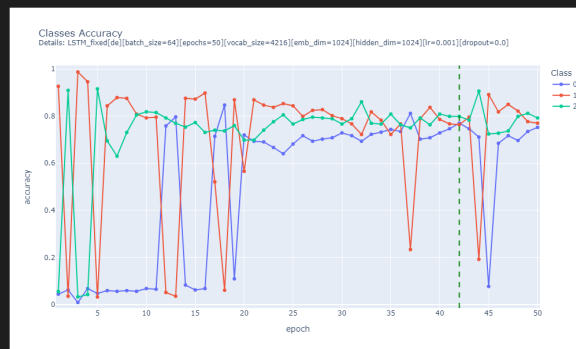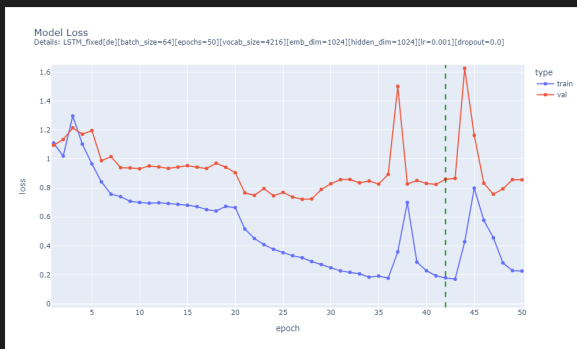
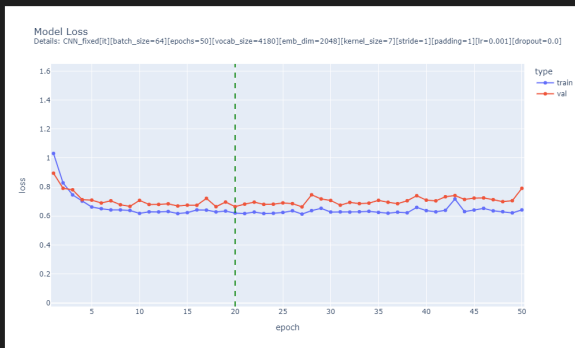## Classes Accuracy

CNN Training Results

Models Loss

Classes Accuracy

En
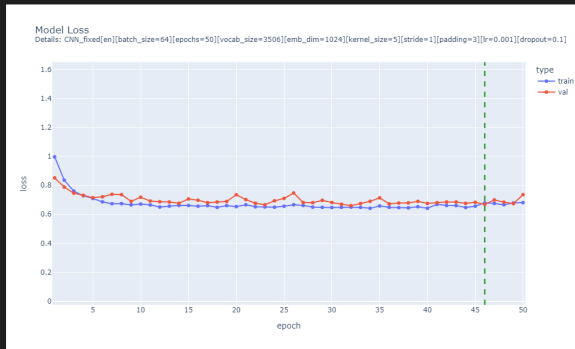
De

It

Pl

Sv
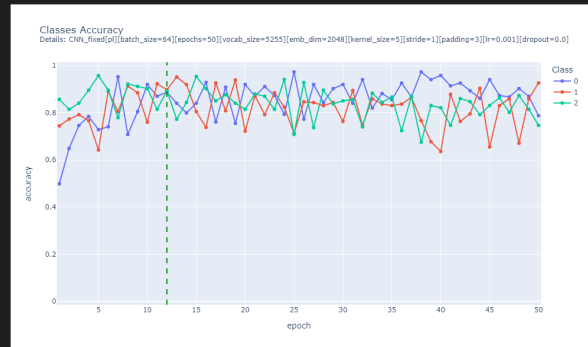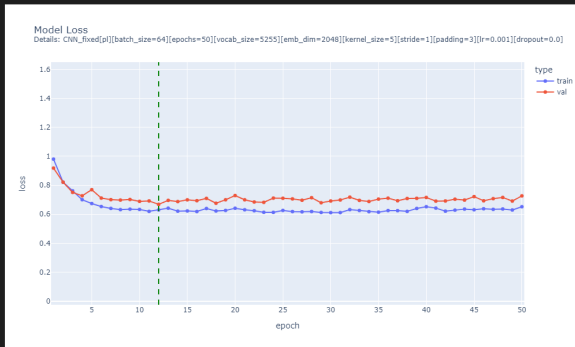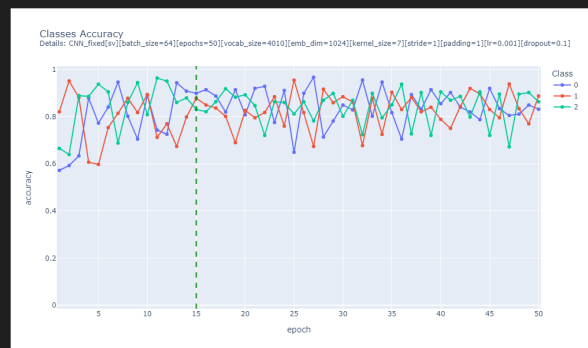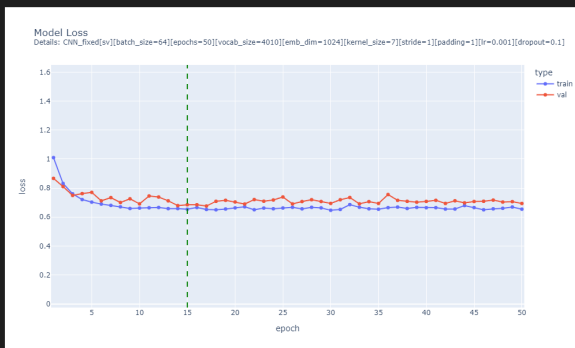
References

[1] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.

[2] Vimala Balakrishnan and Ethel Lloyd-Yemoh. Stemming and lemmatization: a comparison of retrieval performances. 2014.

[3] Jason Brownlee. A Gentle Introduction to Cross-Entropy for Machine Learning.
https://machinelearningmastery.com/cross-entropy-for-machine-learning/.
Accessed: 27-05-2022.

[4] Jason Brownlee. A Gentle Introduction to Long Short-Term Memory Networks by the Experts.
https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/.
Accessed: 23-05-2022.

[5] Jason Brownlee. A Gentle Introduction to Pooling Layers for Convolutional Neural Networks.
https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/.
Accessed: 27-05-2022.

[6] Jason Brownlee. How Do Convolutional Layers Work in Deep Learning Neural Networks?
https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/.
Accessed: 27-05-2022.

[7] Denis Eka Cahyani and Irene Patasik. Performance comparison of tf-idf and word2vec models for emotion text classification. *Bulletin of Electrical Engineering and Informatics*, 10(5):2780–2788, 2021.

[8] Ilias Chalkidis, Manos Fergadiotis, and Ion Androutsopoulos. MultiEURLEX - A multi-lingual and multi-label legal document classification dataset for zero-shot cross-lingual transfer. In *EMNLP*, 2021.

[9] Alex Graves and Juergen Schmidhuber. Framewise phoneme classification with bidirectional lstm networks. *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, 4:2047–2052 vol. 4, 2005.

[10] IBM. Natural Language Processing (NLP).
https://www.ibm.com/cloud/learn/natural-language-processing.
Accessed: 21-05-2022.

[11] Julie Beth Lovins. Development of a stemming algorithm. *Mech. Transl. Comput. Linguistics*, 11(1-2):22–31, 1968.

[12] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5):544–551, 2011.

[13] Tom O'Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. Kerastuner. https://github.com/keras-team/keras-tuner, 2019.

[14] PyTorch. Dropout.
https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html.
Accessed: 24-05-2022.

[15] PyTorch. Embedding.
https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html.
Accessed: 24-05-2022.

[16] PyTorch. Linear.
https://pytorch.org/docs/stable/generated/torch.nn.Linear.html.
Accessed: 24-05-2022.

[17] PyTorch. LSTM.
https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html.
Accessed: 24-05-2022.

[18] Towards Data Science. The Importance of Natural Language Processing for Non-English Languages.
https://towardsdatascience.com/the-importance-of-natural-language-processing-for-non-english-languages-ada463697b9d.
Accessed: 21-05-2022.

[19] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *ArXiv*, abs/1409.2329, 2014.