MASTER PROGRAMME:
«INFORMATION MANAGEMENT IN LIBRARIES, ARCHIVES, MUSEUMS»

ΤΜΗΜΑ ΑΡΧΕΙΟΝΟΜΙΑΣ, ΒΙΒΛΙΟΘΗΚΟΝΟΜΙΑΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΗΣΗΣ
ΣΧΟΛΗ ΔΙΟΙΚΗΤΙΚΩΝ, ΟΙΚΟΝΟΜΙΚΩΝ ΚΑΙ ΚΟΙΝΩΝΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

DEPARTMENT OF ARCHIVAL, LIBRARY AND INFORMATION STUDIES
SCHOOL OF MANAGEMENT, ECONOMICS AND SOCIAL SCIENCES

Dissertation

Deep-Learning vs Classical Machine-Learning comparison for Text Classification

Author:

Ioannis Drizis (206682005)

Supervisor: Ioannis Triantafyllou

Athens, July 2022

Επιτροπή Εξέτασης

1. Ιωάννης Τριανταφύλλου

Ioannis Triantafyllou

Digitally signed by Ioannis Triantafyllou
Date: 2022.07.14 19:54:34 +03'00'

2. Δημήτριος Κουής

Dimitrios Kouis

Digitally signed by Dimitrios Kouis
Date: 2022.07.15 09:18:00 +03'00'

3. Σαράντος Καπιδάκης

[illegible]

Sarantos Kapidakis
2022.07.15
04:00:11 +03'00'

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΜΕΤΑΠΤΥΧΙΑΚΗΣ ΕΡΓΑΣΙΑΣ

Ο κάτωθι υπογεγραμμένος Ιωάννης Δρίζης του Κωνσταντίνου με αριθμό μητρώου 206682005, φοιτητής του Προγράμματος Μεταπτυχιακών Σπουδών «Διαχείριση Πληροφοριών σε Βιβλιοθήκες, Αρχεία, Μουσεία του Τμήματος Αρχειονομίας, Βιβλιοθηκονομίας και Συστημάτων Πληροφόρησης της Σχολής Διοικητικών, Οικονομικών και Κοινωνικών επιστημών του Πανεπιστημίου Δυτικής Αττικής, δηλώνω ότι:

«Είμαι συγγραφέας αυτής της μεταπτυχιακής εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της, είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, οι όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε ακριβώς είτε παραφρασμένες, αναφέρονται στο σύνολό τους, με πλήρη αναφορά στους συγγραφείς, τον εκδοτικό οίκο ή το περιοδικό, συμπεριλαμβανομένων και των πηγών που ενδεχομένως χρησιμοποιήθηκαν από το διαδίκτυο. Επίσης, βεβαιώνω ότι αυτή η εργασία έχει συγγραφεί από μένα αποκλειστικά και αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο δικής μου, όσο και του Ιδρύματος.

Παράβαση της ανωτέρω ακαδημαϊκής μου ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση του πτυχίου μου».

[illegible]

Αθήνα, 01/07/2022

Ιωάννης Δρίζης

Acknowledgements

I would like to thank Professor Ioannis Triantafyllou for giving me the chance to draft this dissertation. The dissertation’s topic allowed me to learn the Python programming language, and to get involved with basic features of Machine Learning, mostly related to Text Classification. This enhanced my skills in understanding and implementing complex coding concepts, which are necessary in most contemporary business needs. Also, I would like to thank the master’s selection committee for accepting my participation in this master. The master’s curriculum provided me with theoretical and practical knowledge related to Information Management in Libraries, Archives and Museums. Modern societies are more and more dependent to Information and Information management, thus, this master boosted my capabilities and helped me to understand better the related concepts.

01/07/2022

Ioannis Drizis

Abstract in English

The Classical Machine Learning and Deep Learning models are used to provide solutions in everyday technologies, like weather prediction, stock price prediction, voice-to-text conversion, fraud detection, quality assurance, etc. These implementations are only a part of a broad range of applications where these algorithms can offer unique services.

In this dissertation, Classical Machine Learning models will be compared with Deep Learning Neural Network models, within the frame of Text Classification. This comparison will be done by using three different feature selection metrics, namely tf.idf, chi square (x^2) and devmax.DF. Also, different Neural Network Deep Learning architectures are tested and compared, as well as different parameters (input vector size, topology architecture, etc.), which are applied in Neural Networks.

Keywords: Machine Learning, Deep Learning, Text Preprocessing, Text Classification, Devmax.DF

Περίληψη (Abstract in Greek)

Η Μηχανική Μάθηση και η Βαθιά Μηχανική Μάθηση, εφαρμόζονται σε τεχνολογίες καθημερινής χρήσης, όπως οι μετεωρολογικές προβλέψεις, η πρόβλεψη της τιμής μιας μετοχής, η μετατροπή ήχου σε κείμενο, η αναγνώριση απάτης, η διασφάλιση ποιότητας, κλπ. Αυτές οι εφαρμογές αποτελούν μονάχα ένα μικρό κομμάτι από το τεράστιο εύρος, όπου εφαρμόζονται μοντέλα Μηχανικής Μάθησης και Βαθιάς Μάθησης.

Σε αυτή την πτυχιακή, θα συγκριθούν κλασσικά μοντέλα Μηχανικής Μάθησης με μοντέλα Βαθιάς Μάθησης, μέσα στο πλαίσιο της Κατηγοριοποίησης Κειμένου. Η σύγκριση θα λάβει χώρα με τη χρήση τριών διαφορετικών μετρικών εξαγωγής χαρακτηριστικών: την tf.idf, χ- τετράγωνο (χ^2) και devmax.DF. Επιπλέον, θα εξεταστούν και θα συγκριθούν διαφορετικές αρχιτεκτονικές Νευρωνικών Δικτύων Βαθιάς Μάθησης, όπως επίσης και διαφορετικές παραμέτρους (μέγεθος διανύσματος εισαγωγής, αρχιτεκτονική τοπολογίας, κλπ.), οι οποίες εφαρμόζονται σε Νευρωνικά Δίκτυα.

Λέξεις Κλειδιά: Μηχανική Μάθηση, Βαθιά Μηχανική Μάθηση, Προεπεξεργασία Κειμένων, Κατηγοριοποίηση Κειμένων, Devmax.DF

Table of Contents

Table of Figures

Table of Matrices

Table of Terms

Application Programming Interface (API)	A connection between computers or computer programs
Artificial Intelligence	The effort to automate intellectual tasks normally performed by humans
Classification algorithms	Algorithms that classify data into categories, also known as "classes"
Deep Learning	A group of algorithms, that belong into the Machine Learning family. Their main difference is that the learning-process is achieve via layers. The number of the layers characterizes the model's "depth"
Dimentionality reduction	A practice that helps reducing complexity of data, without losing too much information
Ensemble Method	A group of Machine Learning models that enhance prediction capabilities
Feature engineering	The art of processing/handling the features in such a way that it will enhance performance
Feature extraction	The techniques of extracting information from the features by combining or processing them
Feature selection	The techniques of selecting the right features that will enhance performance
Machine Learning	A group of algorithms that learn from data and crate rules. These rules are later used to make predictions
Natural Language Processing	An area of computer science related to human language
Neural Network	Most deep learning algorithms refer to the Neuron Networks models
Regression algorithms	Algorithms that predict a value (e.g. future price of market-stocks, temperature, etc.)
Test Set	A dataset used to test Machine Learning models' performance in the final stage
Text Classification	The process of assigning a text to a specific category
Training Set	A dataset used to train Machine Learning models
Validation Set	A dataset used to validate Machine Learning models’ performance in an early stage

Chapter 1. Introduction

Classical Machine Learning models and Deep Learning models are used more and more to everyday applications (Sarker, 2021). Thus, there is an interest of further searching their capabilities and potential. In this dissertation, a comparison of classical Machine Learning models and Deep Learning models will be performed, within the context of Text Classification. Regarding the classical Machine Learning models, the following models will be tested: Naïve Bayes, k-NN, SVM, Random Forest and Logistic Regression. In Deep Learning, different model schemas will be tested, and the results will be compared with those of the Classical Machine Learning tests.

Three different feature selection metrics will be used during this procedure, namely tf.idf, chi square (x^2) and devmax.DF. The latter is a newly introduced feature selection metric for Text Classification, which has produced great results (Triantafyllou, Drivas, & Giannakopoulos, 2020). Specifically, this metric was tested on a dataset that consists mobile application reviews, with the use of classical Machine Learning models. That paper shows that devmax.DF outperformed other feature selection metrics, such as tf.idf and x^2.

Chapter 2. Theory

In this chapter, an overview of the theory is given, regarding Machine Learning (ML) and Text Classification (TC) concepts. This is necessary so the reader can understand the basic concepts of the topic. In general, the relevant literature can be distinguished into two categories. The first one, encompasses bibliography with a more abstract approach of ML and Deep Learning (DL) concepts and terminologies. In the second one, the bibliography is oriented more into Natural Language Processing (NLP) concepts (but withing the framework of ML).

2.1 Machine Learning and Deep Learning

The term Machine Learning is used to describe algorithms that "learn" from data (Geron, 2019, p. 3). The term appeared for the first time in the 50' (Samuel, 1959), but only the last 10-20 years ML has prevailed as a significant filed in computer science. ML is a subcategory of Artificial Intelligence (AI). The latter is defined as the process to automate intellectual tasks normally performed by humans. Thus, AI is considered as a more abstract concept, which includes Machine Learning (Chollet F. , 2017, p. 1).

Deep Learning belongs to the ML family, but it is considered as a subcategory of ML (Goodfellow, Bengio, & Courville, 2016). The different feature in DL is that the learning process is achieved through consecutive layers. Except from the input and output layer, one may insert a handful or even hundreds of layers that in various ways achieve to reach high scores in predictions (Chollet F. , 2017, pp. 6-10). An overview of AI, ML and DL is depicted in Figure 1.

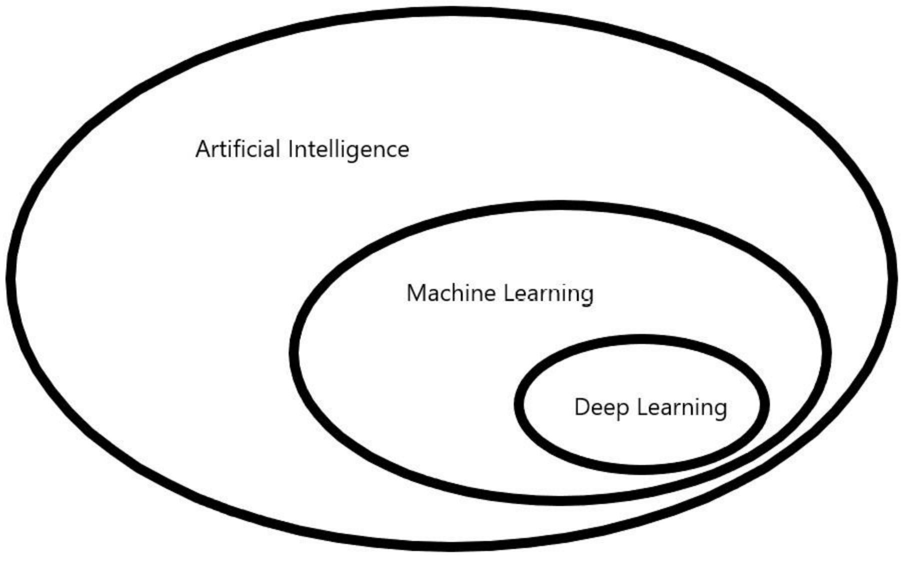


Figure 1: Overview of AI, ML and DL

Machine Learning algorithms “learn” from data and generate predictions based on the learning process. This learn-and-predict process is the key-value in ML, compared to other algorithms like “Symbolic AI”, which is not much more than a huge set of rules implemented into an algorithm (Chollet F. , 2017, p. 3). In ML, the user doesn’t have to create rules. The rules are created via the learning process (Figure 2). So, in general, one may use ML for different tasks, such as Optical Character Recognition (OCR), face recognition, voice recognition, price prediction, spam e-mail detection, etc.

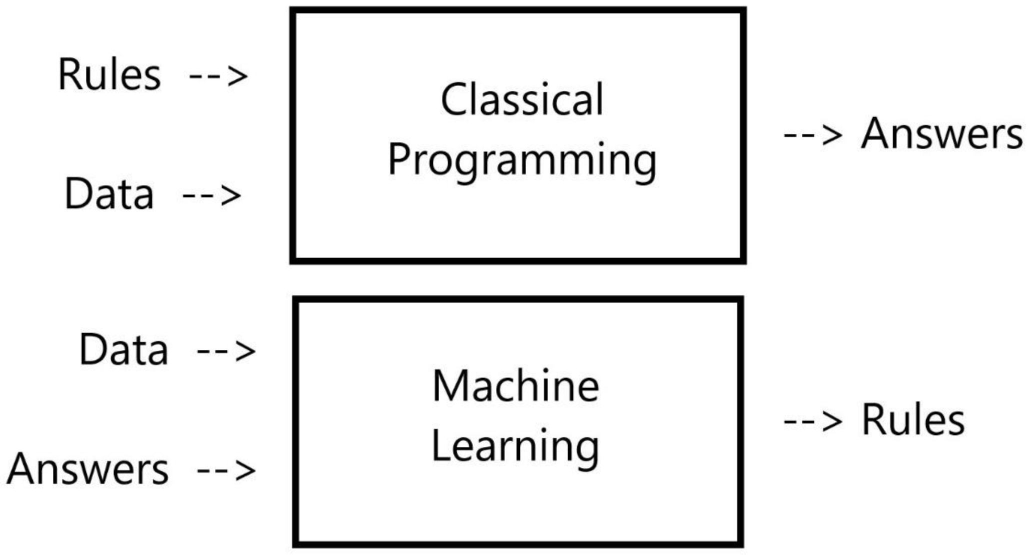


Figure 2: Classical programming vs machine learning

Furthermore, one may distinguish four different “branches” in Machine Learning: Supervised, Unsupervised, Self-Supervised and Reinforcement learning (Chollet F. , 2017, pp. 85-87) as shown in Figure 3. Supervised ML refers to a learning process, in which the data is already labelled (by humans), thus the model tries to figure out what is needed to correctly predict the labelled dataset. Unsupervised ML models cluster the data based on specific algorithms and may provide useful insight into the dataset, such as trends, anomalies, clusters, etc. Unsupervised ML is usually used to handle data this is not labelled. Self-Supervised models are actually supervised models, but the labels are created algorithmically (and not by a human). For example, one may use an Unsupervised ML model to categorize the data, and then implement a Supervised ML model on it. Lastly, Reinforcement Learning (Russell & Norvig, 2016; Sutton & Barto, 2018) is a process, through which the model understands its environment and optimizes actions which lead to a higher score. For the moment, Reinforcement Learning hasn’t been developed as much as the other “branches”, but it has been widely used in the gaming industry.

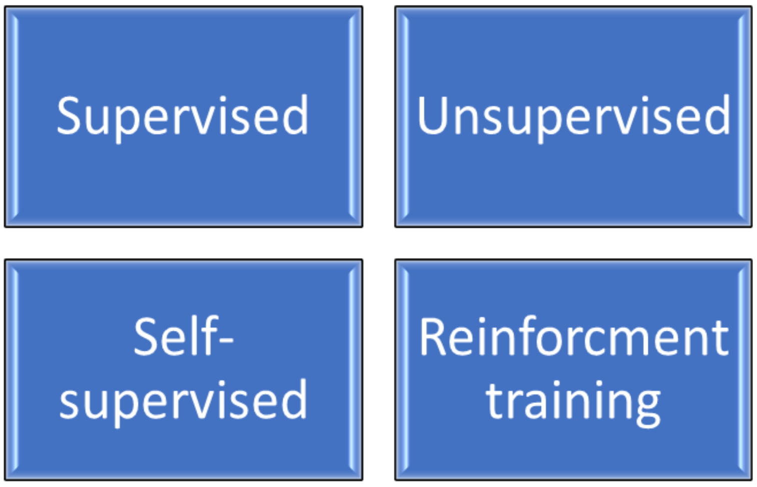


Figure 3: "Branches" of Machine Learning

In this dissertation, the focus will be on the supervised "branch" of Machine Learning.

In addition, the training procedure may take place with two different ways (Geron, 2019, pp. 14-17). The first one is called "batch learning" and the second one "online training". In batch learning the model receives all the data in a one-off procedure. On the other hand, in online training, the model is "fed" with data step by step, so the model "learns" incrementally. This learning procedure is usually chosen whenever there is a continuous flow of data (e.g., stock exchange, cryptocurrency, etc.).

Lastly, another remark related to Machine Learning models, is that their input data should be a list of numbers (Chollet F. , 2017, pp. 93, 295). So, information from real world should somehow be transformed into a list of numbers (e.g., text, pictures and voice should be converted into numbers, etc.). Only if this task is done, can one proceed further.

2.1.1 Classical Machine Learning

There are many ML models. Each of them operates algorithmically in a different way. In this paragraph, you may find a brief introduction into the way some of the classical ML Classifiers "think", without analyzing the mathematical background of their procedure.

Probabilistic models

A handful of Classification models calculate the probability of an entity to belong in a category (Kubat, 2017, pp. 19-42), based on the data provided during the training procedure. The Naive Bayes Classifier (Webb, 2011) and the Logistic Regression Classifier (Hastie, Friedman, & Tibshirani, 2001) belong to this kind of models.

k-Nearest Neighbors (kNN)

The k-Nearest Neighbors Classifier (Altman, 1992) categorizes the data according to the k nearest data points (Kubat, 2017, pp. 43-46). For example, imagine a binary dataset with blue and red labels. A new data point (marked with an asterisk in Figure 4) will be labeled according to the k criteria that were introduced into the Classifier's parameters.

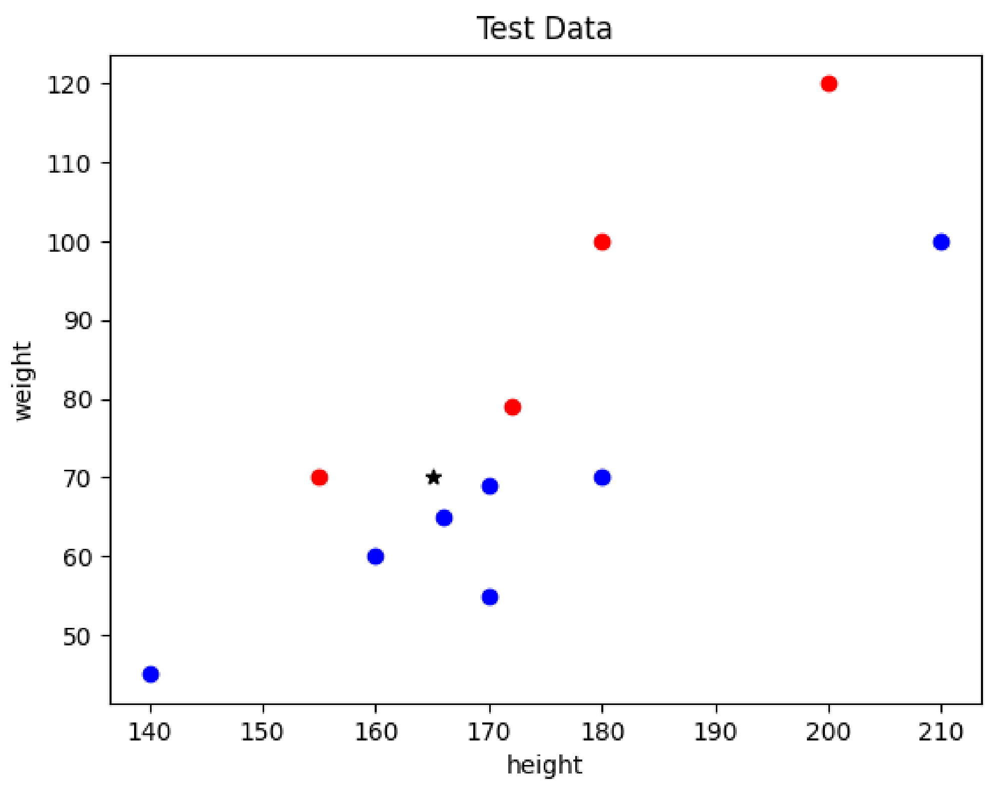


Figure 4: Example of a k-NN Classifier

So, if k=1 the algorithm tries to find the 1 nearest datapoint. In the above Figure 4, it seems that the nearest datapoint is the blue one. In this case, the Classifier will categorize this new entity into the blue class. One may substitute k to any integer, but one should have in mind that it is preferred to assign to k an odd integer. This resolves possible conflict that would trigger with an even k, in case for example k is equal to 2, and one of the nearest neighbors is True (red) and the other one is False (blue). The Classifier won't be able to reach a logical conclusion. Of course, more parameters can be added to that, such as to consider the distance between the two datapoints. In that case, long distance won't be as important as the short distance.

Kernel methods

A group of Classifiers belong to the Kernel methods, such as the Support Vector Machine (SVM), the kernel Fisher discriminant analysis, the kernel principal component analysis (PCA), (Muller, Mika, Ratsch, Tsuda, & Scholkopf, 2001). All of these implement the “kernel trick”, which allows to intergrade a high-level of polynomial features of a dataset, without the need to actually add them, thus saving a lot of calculation resources.

The Support Vector Machine model is the most known kernel model, developed in the late nineties (90’) that provides solutions to linear and non-linear tasks. SVMs map the data into a new high-dimensional area and then create a decision boundary (Chollet F. , 2017, pp. 13-14).

Though SVMs perform very well, they are not preferred for large datasets, due to the high amount of time they need to get trained. Also, they usually need beforehand a good feature extraction technique. SVMs can also be used for regression and for outlier detection (Geron, 2019, p. 153).

When creating an SVM model, a decision boundary is drafted as said before. Sometimes, the dataset is too complex, and it might be useful to “allow” some mistakes while the model is drafting the boundary. This strategy is called “soft margin classification” and it is common to use it in SVMs (Geron, 2019, pp. 154-156). Actually, the model creator allows some margin violations.

When non-linear SVM models are applied, another parameter can be implemented. This is the degree parameter of the model, which is equal to the degree of the equation that is created in order to draft a non-linear parameter (Geron, 2019, pp. 158-159).

Decision Tree – Random Forest

Though Decision Trees (DT) were studied previously (Breiman, Friedman, Olshen, & Stone, 1984), they gained attention in the early 2000 and by the end of that decade they were preferred over the kernel methods. They are easier to explain and to depict. Decision Trees are flowchart-alike and reach a decision by parsing sequences of “if” statements, as shown in Figure 5 (Chollet F. , 2017, pp. 14-15). Decision trees have a parameter called “depth”. The depth is equal to the maximum length of the sequence of “if” statements within the model (Geron, 2019, pp. 176-178). In the example of Figure 5, the model’s depth is equal to two. There is no specific rule, regarding how deep should be a Decision Tree. One may experiment with the data and the model to find the optimum number of the depth parameter.

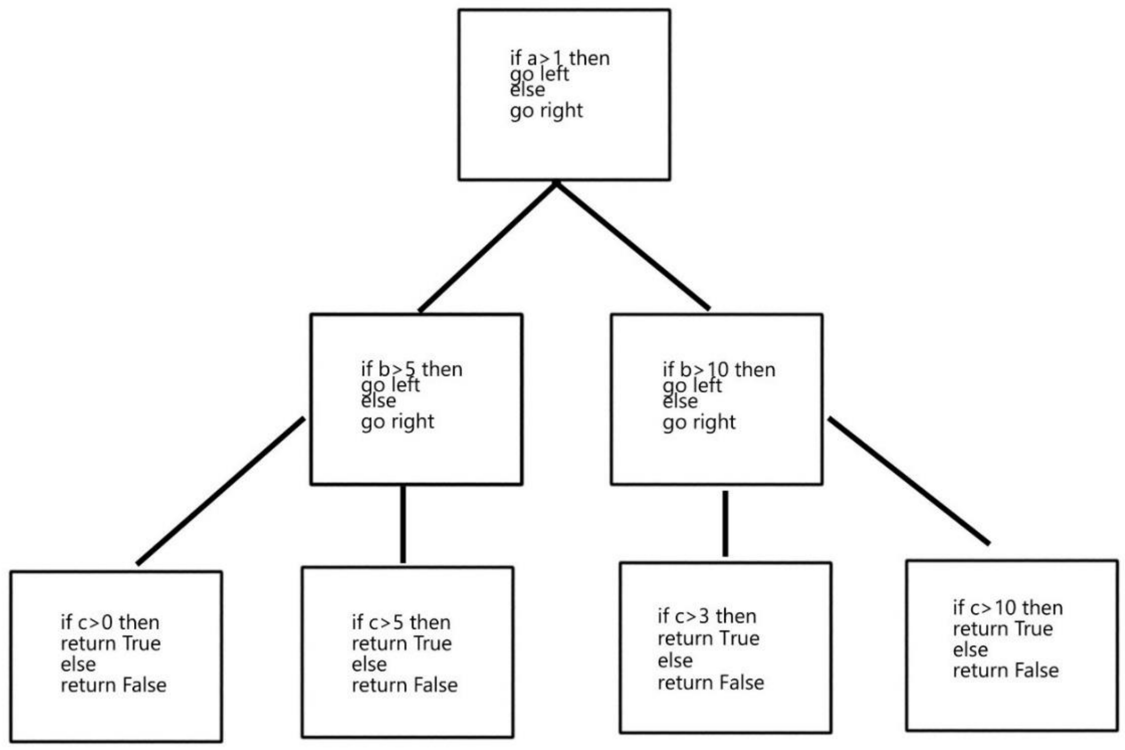


Figure 5: Example of a Decision Tree Classifier

A Random Forest (RF) consist of multiple Decision Trees (Geron, 2019, p. 197; Ho, 1995). Each DT yields a prediction, and in the end the class with the most votes is the “winner”. The procedure of combining different Machine Learning algorithms is called “ensemble” (Geron, 2019, p. 189). So, Random Forest is an ensemble of Decision Trees.

2.1.2 Deep Learning

Deep Learning (DL) belongs to the ML family, as said previously. Alternative names that could be given to this subcategory of Machine learning are “layered representations learning” or “hierarchical representations learning” (Chollet F. , 2017, pp. 6-7), because the learning process is achieved via layers. The number of layers is considered as the “depth” of the model. A few layers means that the model is “shallow”. On the other hand, State of the Art Deep Learning algorithms consist of hundreds of layers or even more. Furthermore, each connection from one layer to the next has its unique weights, which are initially arbitrary, but via the learning process, the weights are adapting to the optimum values. Most DL models are Neural Networks, which “remind” human brain neurons.

In Figure 6, an arbitrary Deep Learning Neural Network model is depicted with an input layer of five values and an output layer of two values. In between there are two layers, which are called “hidden layers”; in this model, each hidden layer consists of three hidden units. Hidden units (Chollet F. , 2017, p. 63) are also known as neurons (Geron, 2019, p. 325) or nodes (Vajjala, Majumder, Gupta, & Surana, 2020, p. 28) and represent a single datapoint. The first value of the input layer is connected with the three hidden units of the first hidden layer (this occurs also for the other input values; the second value of the input layer is also connected with the three hidden units of the first hidden layer, as well as the third value of the input layer is connected with the three hidden units of the first hidden layer, etc.). This means that the information of each input value is transferred to the next layer. Each connection has a weight, which at the beginning is random. The weights play a key-role in the procedure, as these are used to calculate the output from one layer to another, until the final output.

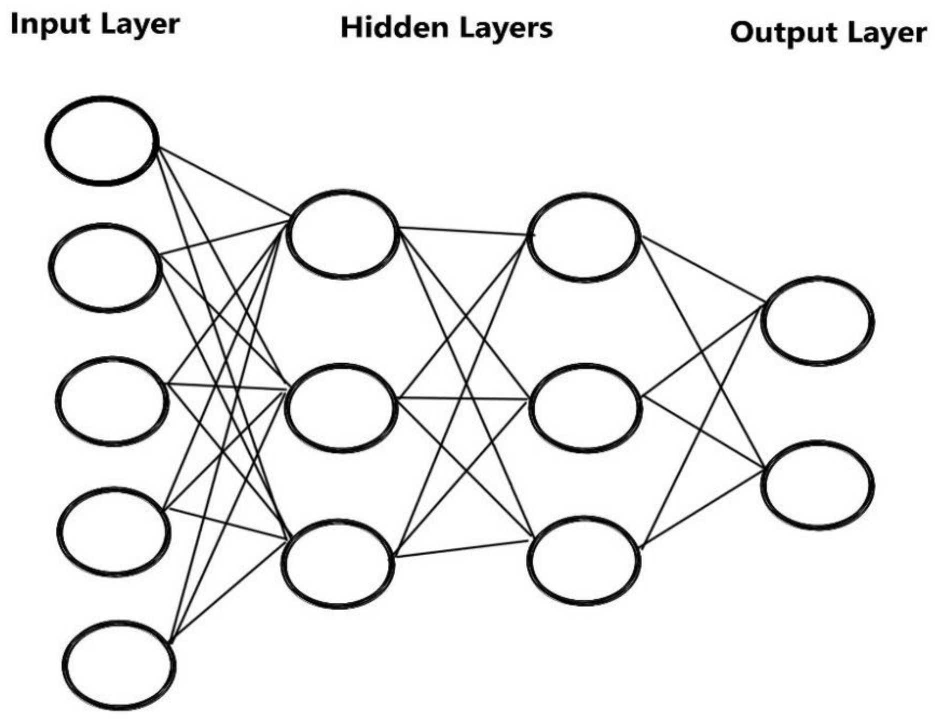


Figure 6: Example of Deep Learning Neural Network layers

Note that another two parameters are important in the creation of a Deep Learning architecture (Chollet F. , 2017, pp. 63-66):

- The way each layer connects with the next one.
  A Dense layer is chosen when the aim is to create a full connection between the layers (and specifically with the preceding layer). A CNN (Convolutional Neural Network) layer is chosen for pattern recognition (e.g. pictures) or sequence understanding that can be used for Text Classification (Chollet F. , 2017, pp. 244-245). An LSTM

(Long-Short Term Memory) layer can also be used for sequence recognition (Chollet F. , 2017, pp. 187, 208).

- The activation function of each layer.
  An activation functions such as “relu” or “softmax”, regulates the flow of data by activating or not a hidden unit and by further passing the value with specific parameters.

Furthermore, Deep Learning models “operate” circle-wise (Figure 7). To completely understand the basic concepts of DL, one should become familiar with DL’s terms such as loss score, loss function, optimizer, batch size and epochs (Chollet F. , 2017, pp. 7-9). During the learning process Deep Learning algorithms predict a value, which is the model’s outcome. The outcome is compared with the actual value. This comparison results to a “loss score” (the “distance” between the actual value and the predicted value). The loss score is calculated thanks to a functionality of Deep Learning named “loss function”. The loss score is used to re-calculate the layers’ weights, thanks to another functionality of DL: the “optimizer”. The learning process is achieved by running the learning process n-times. Each repetition is considered as one “epoch”. One must find the ideal epochs for a specific DL model (not too few, nor too many, so to avoid overfitting or underfitting)[1]. In each epoch the weights adapt more and more to the data, and by this way the model becomes more successful in predictions. Finally, one may add the batch size parameter into the Deep Learning model (Geron, 2019, p. 326). The batch size refers to the amount of the instances that are trained in an epoch. For example, a dataset with 1,000 instances and a DL model with a training batch size of 100 will be trained in one epoch 1,000/100 = 10 times. The main difference between epochs and batch size, is that the epoch refers to the training procedure in the whole dataset, whereas the batch size refers to the chunks of the data that is trained in one epoch.

[1]In page 16 you can find more details regarding overfitting and underfitting.

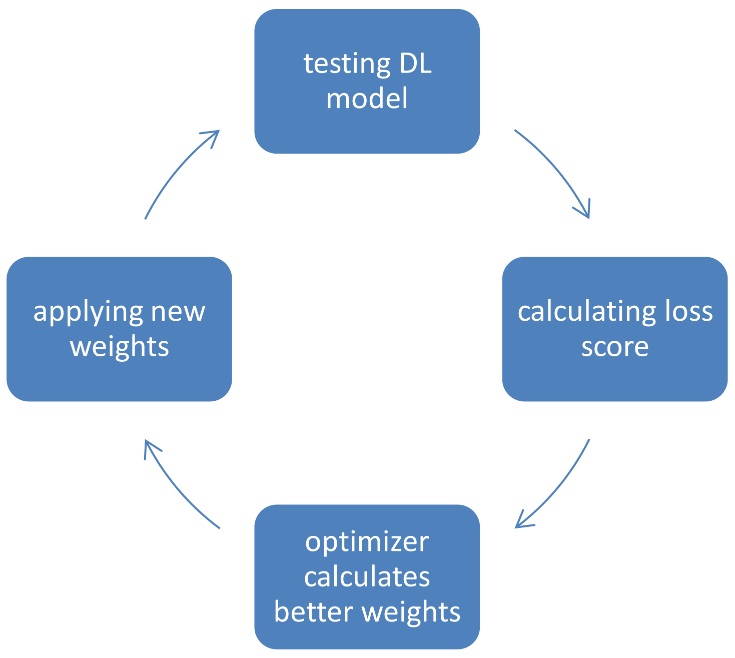


Figure 7: The Learning Process of Deep Learning

Further to the above, DL models may have more parameters to regulate overfitting or underfitting, such as one or more Dropout layers, and/or the Early Stopping feature (Geron, 2019, pp. 141, 365). The Dropout layer may span between 0% and 100%. This represents the chance of a hidden unit, to be ignored during training. Interestingly, this enhances performance. Some use a combination of models that were trained with Dropout layers. This is called "Monte Carlo" Dropout, which increases performance. On the other hand, the Early Stopping feature is also used during training to prevent overfitting. Early stopping allows the model to stop further training when there is no performance improvement, thus, to avoid the model's adaptation to specific training-data.

As it is noticed, the creation of a Deep Learning architecture involves thinking of:

- How many layers should be inserted, and of what length?
- Which loss function shall be used?
- Which optimizer shall be used?
- What other parameters should be introduced?

Francois Chollet (2017, p. 54) claims that creating a Deep Learning structure is more Art than Science. Since implementing Deep Learning is mostly an experimental field and lacks theory (Chollet F. , 2017, pp. 17-18), could one argue with this claim?

In general, DL models are considered as the best solution for big volumes of data. There is no high demand of complex feature engineering -some argue that there is no need for feature engineering at all in Text Classification (Vajjala, Majumder, Gupta, & Surana, 2020, p. 62)-, plus there is a worldwide trend in moving to the Deep Learning model prediction, due to its efficiency. For example, CERN has switched from Decision Trees and SVM to Deep Learning models (Chollet F. , 2017, p. 16).

2.1.3 Types of Predictions

Machine Learning and Deep Learning models can predict a value as part of a regression task (Chollet F. , 2017, p. 78) or may predict the class in which an instance of data should be categorized, which is called “classification task”.

Predicting a value as part of a regression task is useful to estimate house prices, stock-prices, temperature of the atmosphere for weather forecasting, etc. and has no further complexity in describing it. On the other hand, classification problems vary and can be further distinguished to “binary”, “multi-class”, “multi-label” and “multi-output” (Geron, 2019, pp. 88, 100, 106-108).

Binary classification models are used when the data is split between only two categories. Some ML models, such as Logistic Regression and Support Vector Machine classifiers can perform only binary classification.

Multi-class classification models learn to distinguish data between n-categories. Such classifiers can be used for Optical Character Recognition (OCR) to determine if a character belongs to the range of characters: “0-9”. In this case, a character cannot belong to more than one category, but the model must decide to which category it belongs.

Multi-label classification models are used when the dataset can have more than one label. For example, news articles can have multiple labels simultaneously. These classifiers can label an instance with more than one class.

Multi-output classification models are like multi-label classification models, but they can handle multiple classes for each label.

2.1.4 Evaluating Machine Learning models

There is at least a handful of metrics and features that are used to evaluate the efficiency of Machine Learning models, such as accuracy, the confusion matrix, precision, recall, the F_1-score (Geron, 2019, pp. 88-100) and the ROC curve (Fawcett, 2006). These metrics provide feedback on how successful a model is in predicting the right class. For regression tasks there are other metrics, such as calculating the distance between actual and predicted value.

Nevertheless, in this dissertation the focus will be on a Classification task, thus, in the rest of this paragraph, only the main classification metrics are demonstrated.

Accuracy

Accuracy provides information on how accurate the predictions are. It is calculated by dividing the correctly predicted values to the total sum of dataset (total predictions), as shown in equation 1.

$$\text{accuracy} = \frac{correct\ predictions}{total\ predictions} \qquad \textbf{(1)}$$

Confusion Matrix

The Confusion Matrix provides a better insight in model performance evaluation, because it depicts the outcome in a more detailed way. Particularly, it shows the model’s predictions, and which of these were correctly or incorrectly predicted. The table contains values of the True Positive, True Negative, False Positive and False Negative predictions (Table 1).

Table 1: Example of Confusion Matrix

		Predicted	
		Positive	Negative
Actually	Positive	**True Positive**	**False Negative**
	Negative	**False Positive**	**True Negative**

Precision

The Precision metric is equal to the True Positive value divided by the sum of True and False Positive values (Equation 2). If there is no False Positive value, then the precision scores a 100%.

$$precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \qquad \textbf{(2)}$$

Recall

Recall, also known as Sensitivity or True Positive Rate (TPR) is equal to the True Positive Value divided by the sum of True Positive and False Negative values (Equation 3). If there is no False Negative value, then the recall scores a 100%.

$$recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \qquad \textbf{(3)}$$

F_1-score

The F_1-score is equal to the harmonic mean of Precision and Recall (Equation 4). It is widely used because it includes information for both False Positive and False Negative values.

$$F_1\text{-}score = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 * \frac{precision * recall}{precision + recall} = \frac{TP}{TP + \frac{FN+FP}{2}} \quad (4)$$

Note that both Precision and Recall (and consequently F_1-score) are mainly affected by the True Positive instances. If the dataset has only a few TP instances, a False Negative or a False Positive will strongly affect the results.

Training Set – Validation Set – Test Set

Machine Learning models are trained with data. By training a model with all the dataset's instances (data rows) will not provide the appropriate information regarding its performance.

To tackle this issue, Machine Learning Specialists split the dataset into "Training Set" – "Validation Set" – "Test Set". The goal is to train the model with the training set and to initially evaluate-validate the model with the validation set. Lastly, the test set is used to perform a final check regarding the model's performance (Chollet F. , 2017, pp. 89-92), before implementing the model into the real-world business task.

There are two major types of picking up and implementing the training-validation set "rule". The first one is named "simple hold-out validation" and the second one "k-fold cross validation" (Berrar, 2018). The simple hold-out validation split is an approach, though which the dataset is split only into two parts. One Training Set and one Validation Set (Figure 8). The model is trained with data from the training set, and its performance is calculated with the data of the validation set.

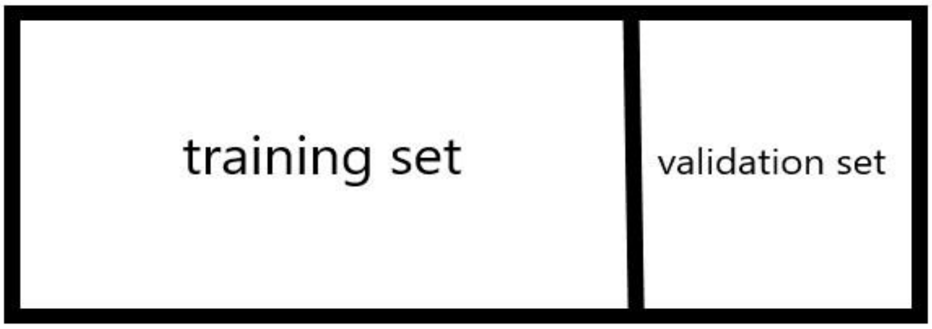


Figure 8: A simple training-validation set example

In the k-fold cross validation split, the dataset is split into k parts. The model is trained and evaluated k-times by using the k-1 instances as training data and the kth part as validation set (Figure 9).

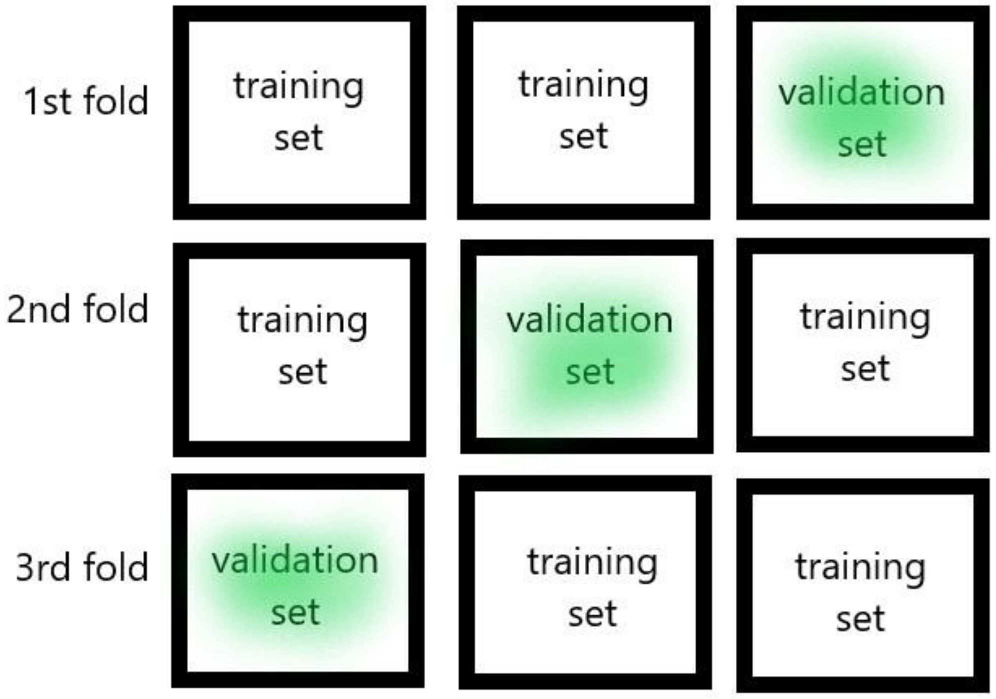


Figure 9: A k-folds cross validation example

The model loops through the folds until every fold is used. For each loop the score of accuracy, recall and precision is calculated. These scores are used in the end to estimate their average. By this way the Machine Learning Specialist may evaluate the model and if needed he can run again the procedure with different parameters in order to reach higher scores.

Another goal of splitting the data into train-validation-test set is because a huge dataset may consist of millions of rows of data. It is time-consuming to use it all and to train a model with such a huge dataset, thus it is preferred to keep a portion of the dataset as training-validation set and to test the models' performance via these sets.

A common approach in splitting the dataset into training and test set is to split it in an analogy of 80% - 20%. This may take place in small and medium size datasets. For bigger datasets, one has to estimate the ideal volume of the training set. As said before, it is difficult to train a model with a dataset that consists of millions of rows of data. Thus, one should slice it and train the model with a smaller proportion. There are no specific rules; the volume of the training dataset depends mostly on the Machine Learning Specialist's intuition.

2.1.5 General Machine Learning Issues

Machine Learning might fail to produce the excellent results, or even a moderate result if some basic factors are not taken into account (Geron, 2019, pp. 23-30). There are two main reasons why a Machine Learning model may fail in scoring high. The first one is a "bad algorithm" and the second one is "bad data".

First, the training set might be too small. Machine Learning is a data-driven field; thus, a small volume of training set wouldn't be able to yield good results. In Figure 10[2] you may see an abstract view on the correlation between ML Performance and Amount of data.

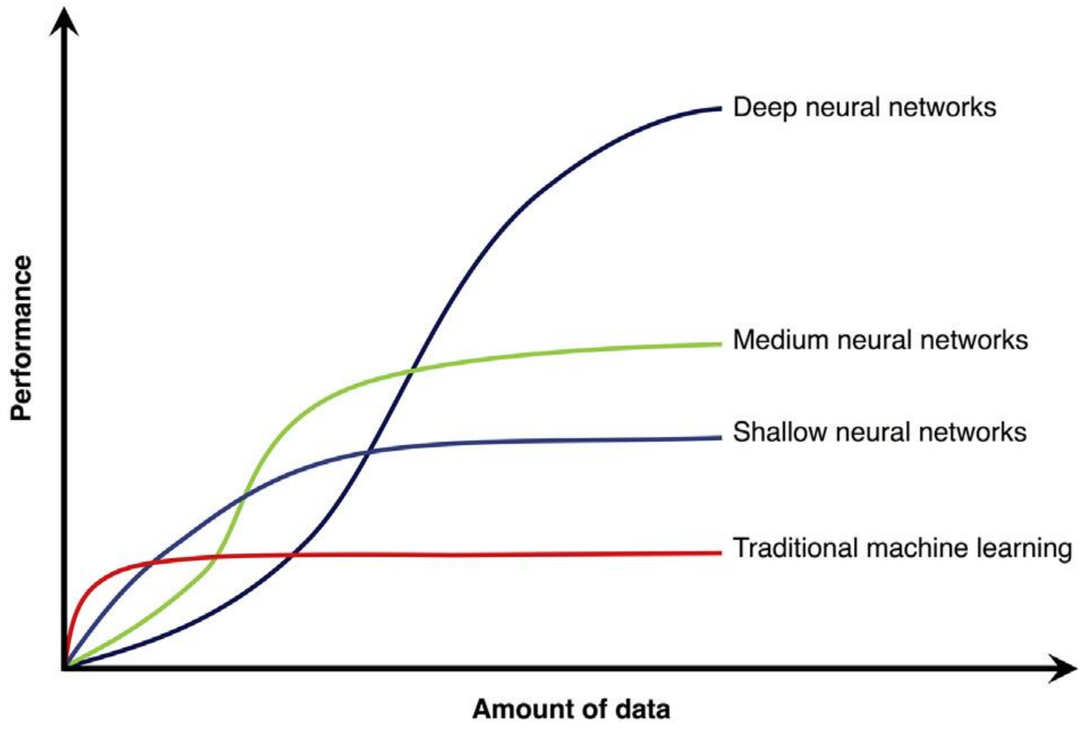


Figure 10: Performance vs Amount of Data in Machine Learning

Secondly, the training data might not reflect the general picture of the whole dataset. In particular, the training set should be proportionate to the dataset. One should not deliberately or by accident create a training set, which has not the right analogies of the whole dataset.

Furthermore, poor-quality data might affect the results. Data with poor quality might involve data entirely irrelevant to the task or data that contain too many missing values. One should

[2] Figure received from paper "Canadian Association of Radiologists White Paper on Artificial Intelligence in Radiology" (2018), reference within paper:

Bahnsen AC. Easy Solutions, Inc. Building AI applications using deep learning. Available at: http://blog.easysol.net/building-ai-applications. Accessed January 28, 2018.

assess the dataset prior using it and accordingly disregard it entirely or handle possible errors or missing values.

Another problematic issue that might affect the models' performance is a low-quality of feature selection or feature extraction method. Machine Learning models use data to learn patterns and create rules. If they are not given the right data and in the best format, they cannot produce the optimum.

Lastly, it is important to mention that a model may overfit or underfit to the training dataset. Both result to low performance. The first one (overfitting) occurs when the model fits too much to the training set (high scores); but so much that it cannot reach high scores when new data is introduced. Literally, the model adapts too much to the data of the training set, that it won't recognize new data easily. The second one (underfitting) occurs when the model's parameters are wrongly set, resulting a model that can't reach high scores nor in the training set, neither in the test set.

2.2. Pre-Processing for Text Classification

Text Classification is the procedure of categorizing text into specific classes. It is considered as one of the most popular features in the field of Natural Language Processing (NLP). Text Classification is also known as "topic classification", "text categorization" or "document categorization" and its applications vary in many fields, such as "Content Classification and organization", "Customer Support", "E-commerce", "language identification", etc. (Vajjala, Majumder, Gupta, & Surana, 2020, pp. 119-123).

The procedure for creating Text Classification Systems could be described as this:

1. Collect and pre-process the dataset
2. Split the dataset into training and test set
3. Transform text into vectors
4. Apply and train a Classifier
5. Evaluate the results
6. Deploy the model into new data and evaluate its performance.

In the following paragraphs, you may find more details related to Text Classification. This will help the reader to understand the basic concepts of the topic.

2.2.1 Text Pre-processing

In Text Classification the dataset consists of letters, words, paragraphs, punctuation, etc. One should pre-process the text, so it shall become easier to analyse it further and to perform feature engineering in a later stage, in the most efficient way. There is a handful of pre-processing practices which are described below (Vajjala, Majumder, Gupta, & Surana, 2020, p. 49).

Word Tokenization

Word Tokenization refers to the method of splitting the text into pieces (Vajjala, Majumder, Gupta, & Surana, 2020, pp. 51-92). Tokenization can be as simple as just to split the text whenever a white space is found, or as complex as to simultaneously remove unnecessary punctuation and/or to ignore numeric values. Of course, in other languages this can be more challenging, like in Chinese language where there is no “space” between words as it is perceived in the English language (Webster & Kit, 1992). By tokenizing the text, the user receives a list of words, which can be further exploited.

Lowercasing - Uppercasing

It is common to lowercase or uppercase all the dataset’s words, in an early stage (Vajjala, Majumder, Gupta, & Surana, 2020, pp. 52-53). By this way the dataset is homogenized and possible issues with comparison of case sensitive programming mechanisms is solved.

Stemming – Lemmatization

Words may have suffixes and/or may have the same lemma with other words. Sometimes, it is useful to “cut” the words’ suffixes or to find the words’ lemma (Jivani, 2011). Particularly, the stemming method removes the words’ suffixes, and the lemmatization method returns the word’s lemma. This procedure reduces the dataset’s variation of features, which in return makes it easier to further analyse the data.

Removing stop-words

Some words do not offer any special information regarding the Classification needs. In the English language, words like “a”, “the”, “of”, etc. are named “stop words” and they are usually removed from the dataset (Vajjala, Majumder, Gupta, & Surana, 2020, pp. 52-53). One may find complex techniques to remove stop-words in the literature (Ladani & Desai, 2020; Gunasekara & Haddela, 2018; Kaur & Buttar, 2018).

Once the text is tokenized, stemmed, lemmatized, lowercased (or uppercased) and once the stop words are removed, the user has created a list of words, which contains all the features, and from which one may extract useful information and train the Machine Learning model (Lane, Howard, & Hapke, 2019, pp. 71-76). This can be named as the corpus Dictionary, Lexicon or Bag of Words.

2.2.2 Feature Selection – Feature Extraction - Vectorizing

Machine Learning performance is dependent on the features, which are selected or/and extracted from a dataset. In the case of Text Classification, one has to select the optimal features from text and create a vector with features that will “feed” the ML model. In Deep Learning, there is an opinion that feature selection is not needed in Text Classification (Vajjala, Majumder, Gupta, & Surana, 2020, pp. 60-61), because DL has the ability to understand the features and to take into account the significant and to ignore the insignificant ones.

Selecting or transforming the dataset’s features is named feature engineering. There are three types of feature engineering practices (Geron, 2019, p. 27). Firstly, one may select specific features from a dataset (feature selection). Secondly, one may produce or combine features by exploiting the already gathered features of a dataset (feature extraction) and lastly sometimes it is needed to search for new data and find new features.

Text consists of letters, words, and punctuation, so one has to decide what should be kept or not from the dataset and what can be further exploited. There are different techniques/metrics which may allow the selection of the words that will be used in the creation of the vector and in the training and implementation of the Machine Learning model, such as “tf-idf”, “chi square” and “devmax.DF”.

Tf.idf

The metric Term Frequency-Inverse Document Frequency (tf-idf, Equation 5) is used to highlight the importance of a term within a document/corpus (Vajjala, Majumder, Gupta, & Surana, 2020, pp. 90-92).

$$(tf\text{-}idf)_k = tf * idf \tag{5}$$

where tf (Equation 6) is the term frequency of the term k:

$$tf_k = \frac{Total\ sum\ of\ occurrences\ of\ term\ k\ in\ document}{Total\ sum\ of\ terms\ in\ document} \tag{6}$$

and idf (Equation 7) the inverse document frequency of term k:

$$idf_k = log \frac{Total\ sum\ of\ Documents\ in\ corpus}{Sum\ of\ documents\ with\ term\ k\ in\ them} \tag{7}$$

Chi square – x²

Chi square (Equation 8) is another metric that is used for feature extraction in Text Classification (Triantafyllou, Drivas, & Giannakopoulos, 2020, p. 8).

$$x^2 = \sum_{i=1}^{c} \frac{(O_i - E_i)^2}{E_i} \tag{8}$$

where $O_i = DF_i$ and $E_i = \frac{DF}{c}$ (DF is the sum of the instances that contain the term F in class i and c is the sum of the classes).

Devmax.DF

Devmax-df (Equation 9) is a newly introduced feature extraction metric that can be used in Text Classification (Triantafyllou, Drivas, & Giannakopoulos, 2020, pp. 8-9).

$$Devmax.df = \frac{\sqrt{\frac{1}{c-1}\sum_{i=1}^{c}(\frac{DF_i}{D_i} - max)^2}}{max} * \log{(DF)} \tag{9}$$

Where $\max = maximum_{i=1}^{c} \frac{DF_i}{D_i}$, c is equal with the sum of the classes, DF is the sum of the instances that contain the term F in class i and D the sum of documents for class i.

Data into Vector

After calculating the metrics, one may extract n-features with the highest score. For each metric there will be a differentiation in the features that were extracted (Table 2). One can experiment on how each metric performs and so decide which fits best to his ML model.

Table 2: Lexicon example

Tf-idf	X^2	Dev.max
Word1	Word2	Word2
Word2	Word5	Word12
Word3	Word8	Word13
Word4	Word1	Word1
Wrod5	Word9	Word5
Word6	Word10	Word10
Word7	Word11	Word4

Notice that in the above arbitrary table, some words are unique in their column, and some are common in two or three columns. Different feature extraction metrics produce different results.

The features which are extracted will be further used to transform each text/document into a vector as described in the following tables. When creating a vector, the user may choose different approaches of calculating the instances of a word within a text. Here two approaches are presented. In the first approach, each word that is found within the text is marked with the number one (1), even if this word can be found more than once in the text (Table 3). This method is called one-hot encoding (Vajjala, Majumder, Gupta, & Surana, 2020, p. 85). Alternatively, the vector may depict all the words' occurrences in the text (Lane, Howard, & Hapke, 2019, pp. 71-76). In that case the vector will be conveying the information about the occurrences into the model, which is actually a bare metric of the document's term frequency (Table 4).

Table 3: One-hot encoding example

	Word1	Word2	Word3	Word4	Word5	Word6	Word7	Word8	Word9
Word1	1	0	0	0	0	0	0	0	0
Word2	0	1	0	0	0	0	0	0	0
Word3	0	0	1	0	0	0	0	0	0
Word4	0	0	0	1	0	0	0	0	0

Table 4: Vector with Bag of Words

	Word1	Word2	Word3	Word4	Word5	Word6	Word7	Word8	Word9
Text1	10	0	0	5	0	0	3	1	15
Text2	0	0	2	4	0	0	4	0	0
Text3	3	2	10	5	0	0	0	0	0
Text4	0	0	0	0	3	1	3	0	0

Once the text is transformed into numeric vectors, the data shall be trained with one or more Machine Learning models. The models may be further used in new documents for evaluation. The one with the best evaluation scores shall be considered as the "winner" of the text classification task. Of course, it is always suggested to re-arrange the models' parameters, so to tackle overfitting or underfitting. Also, one has to slightly change some of the parameters so to optimize the models' prediction capabilities.

Chapter 3. Methodology

Implementing Machine Learning from theory into practice is not always easy. One may find proprietary or open-source tools that provide user friendly solutions (North, 2012). Alternately, it is possible to create "hand-crafted" algorithms and/or to exploit powerful programming language libraries. On both occasions, to reach safe conclusions, one has to experiment with different parameters (Langey, 1988).

In this paragraph, the methodology will be analyzed and specifically:

- the dataset that is used
- the resources (hardware and software)
- the preprocessing procedure
- the feature engineering practices
- the configurations of Classical Machine Learning and Deep Learning models' parameters

Dataset

The same dataset that was used in paper "*How to Utilize my App Reviews? A Novel Topics Extraction Machine Learning Schema for Strategic Business Purposes*" (Triantafyllou, Drivas, & Giannakopoulos, 2020) is used[3]. This dataset consists of reviews of mobile applications. It contains 7,754 reviews, which are categorized between twelve (12) classes. A lot of reviews do not belong to any class (3,713 belong to at least one class – 4,040 do not belong to any class). Nevertheless, they were kept within the dataset as was done in the above-mentioned paper, so to ease comparison with it. The distribution of the classes is shown in Figure 11. The mean per category is equal to 492.25 and you can notice that the dataset in imbalanced.

[3] Retrieved from:

https://github.com/panichella/UserReviewReference-Replication-Package/tree/URR-v1.0

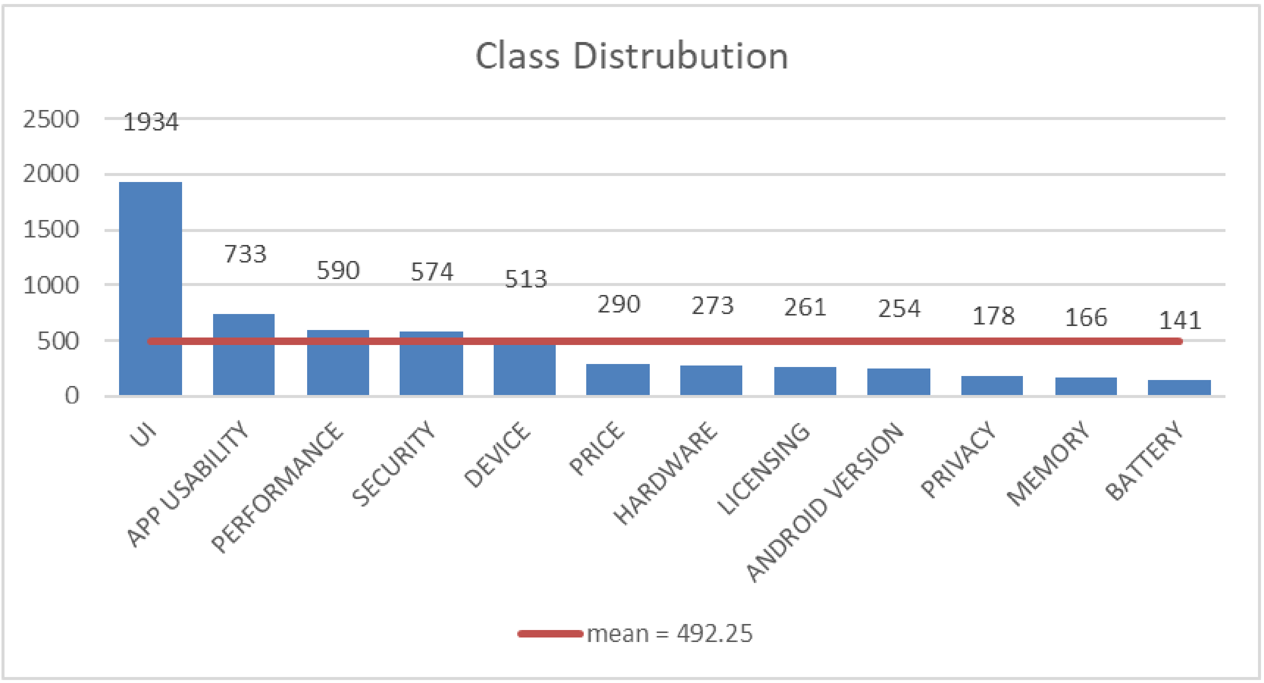


Figure 11: Class distribution of dataset used in ML and DL

The reviews may belong to more than one class; this means that the Classification Task is characterized as “Multi-label Classification”.

Software Resources

To perform all the necessary tasks related to the Classification assignment, the Python[4] programming language (version 3.9) is used via JetBrains’s Free Community user IDE PyCharm[5], alongside with useful programming libraries such as:

- Google’s TensorFlow[6] (Abadi, Agarwal, Barham, & others, 2016) and TensorFlow’s Keras[7] API (Chollet & others, 2015), which exploit’s computer GPU and allows together with Nvidia’s CUDA toolkit and its libraries to run faster Deep Learning models up to five times.

[4] https://www.python.org/

[5] https://www.jetbrains.com/pycharm/

[6] https://www.tensorflow.org/learn

[7] https://keras.io/

- Scikit-learn[8] (sklearn) which is a powerful library (Buitinck & others, 2013) with a wide range of tools for classical Machine Learning and Deep Learning tasks (Pedregosa & others, 2011).
- NumPy[9], which is a powerful framework for scientific computing in Python (Harris, Millman, van der Walt, & others, 2020).
- NLTK[10] (Natural Language Toolkit), which is widely used for Natural Language Processing (NLP) (Bird, Steven, Loper, & Klein, 2009).

Hardware Resources

The programme was initially run (results of paragraph 4.1) with an Asus laptop with the following characteristics:

- CPU: Intel Core i7-7700HQ CPU @ 2.80 Ghz
- RAM: 16 GB
- GPU: Nvidia GeForce GTX 1050, 2781 MB
- Windows 10, 64-bit

Cloud Resources

Results of paragraph 4.2 were produced by running the Python script in Google's Colab cloud service (Bisong, 2019), where one can upload and run Python scripts offline and save files, such as results, directly into user's Google Drive. The cloud service was chosen to boost execution speed and to exploit higher amounts of GPU memory, when computer's memory is not enough to run the Machine Learning scripts. Moreover, via Colab, one may use TPU[11] processors, which are designed specifically for Deep Learning.

Pipeline

The flow of the procedure is drafted and implemented in such a way that it can be run one-off and within loops (each loop has different parameters). Every step is connected with the next

[8] https://scikit-learn.org/stable/

[9] https://numpy.org/

[10] https://www.nltk.org/

[11] Tensor Processing Units

one. This procedure saved a lot of time in running the dataset with different parameters. The pipeline can be described in a very abstract way as shown in Figure 12.

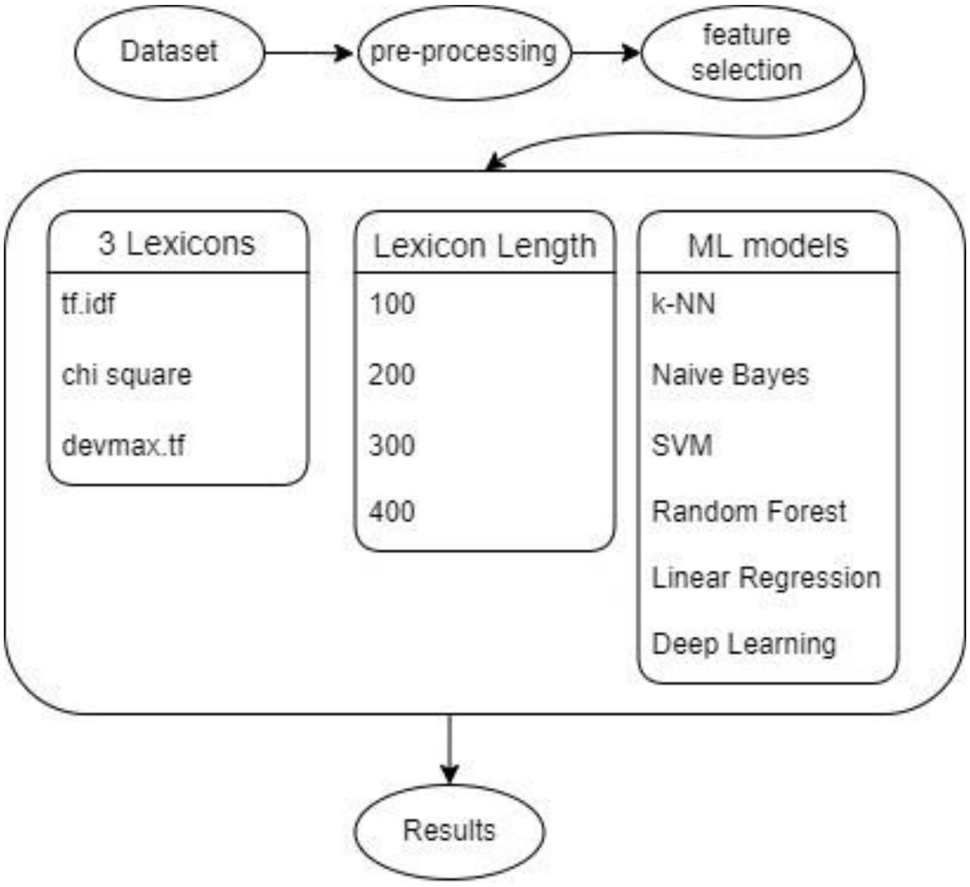


Figure 12: Flow-chart of the Text Classification training pipeline

The script performs three big loops; one through the 3 Lexicons, one through the Lexicons' length parameters and one through the ML models. Each repetition runs seventy-two times (3*4*6=72) and yields 72 results. Each result reflects the models' performance with parameters of Lexicon, Lexicon length and ML model.

Each one-off run has stable parameters of pre-processing methods, feature selection methods and Machine Learning Parameters.

Pre-processing

The dataset was pre-processed with the use of tokenization, lowercasing, and stemming methods. Also, stop words were removed. The pre-processing procedure was achieved via the NLTK library for Python programming language. More specifically, tokenization was performed by using Regular Expression method of NLTK (RegexpTokenizer), with the following regex pattern:

```
\w+(?:-\w+)*
```

The above-mentioned pattern, groups all consecutive characters (letters and numbers), even if they are connected with a hyphen “-“.

The method PorterStemmer of the NLTK programming library is used to stem words.

The lemmatization method was not used, so to distinguish words such as “good”, “better” and “best”. Lemmatization would transform the words “best” and “better” into “good”.

The above procedure produced a list of words. These words were further grouped based on their similarity. More specifically, words with length of more than 4 characters were grouped together, provided that 2/3 of their average length -average(length(word1), length(word2))- is identical.

In the Appendix you may find the code in the Python programming language. More specifically:

- In Appendix – C you may find the code that understands if two words are similar.
- In Appendix – D you may find the code that is used for data preprocessing.
- In Appendix - E you may find the code that creates a Lexicon with all corpus words.

Feature selection

After pre-processing the reviews, a list of words (and group of words) is created much alike a Lexicon. Each word has a different distribution among the whole corpus of the reviews, within each review and amongst the classes. The unique words extracted from the dataset by implementing the pre-processing steps count 6,963 (total words: 100,820 -after using stemming-). After grouping the words, a list of 5,559 records is created (unique and grouped words). So, eventually 6,963-5,559 = 1,404 words were grouped into different groups. This number may vary with other parameters during the pre-processing procedure. Different pre-processing procedure parameters will yield different results most likely.

In this dissertation the metrics of tf.idf, x^2 and devmax.DF are used to select the features with the n-highest score, and specifically the first 100, 200, 300 and 400 words with the highest score of each of the three Lexicons.

Though in DL, it is argued that feature selection is not needed (Vajjala, Majumder, Gupta, & Surana, 2020, p. 60), the selected features will be further used also in DL. This will allow comparison by using the same pre-processing and modeling parameters.

Data into ML models (Vectorizing)

Once the Lexicons are created, the reviews are transformed into vectors (lists of numbers), by implementing the Bag of Words method, by not using frequency counter, but 0 and 1 instead.

The algorithm searches into each review for a word that is within the Lexicon. If found, it appends the value one (1) in the corresponding position of the word in the Lexicon. By default, all vectors' values are given the zero value (0). In the end, each review corresponds to a numeric vector with zero-one values (0-1). This vector is the input value for the Machine Learning and Deep Learning modes.

ML and DL models' parameters

Initially, the dataset is introduced into the ML and DL models with the use of Scikit-Learn library. The following parameters were set for each of the Machine Learning Models:

- Naïve Bayes: the multinominal Naïve Bayes was chosen
- k-NN: the parameter of k=1 was chosen
- SVM: kernel was set to "linear", C[12]=1, degree=3
- Random Forest: the depth was set equal to 15
- Logistic Regression: no parameters
- Deep Learning: The "relu" activation function was chosen; the "adam" optimizer was chosen; a batch size of thirty-three (33) was set and a hidden layer with one hundred (100) hidden units was incorporated; early stopping was activated that stops training when validation score is not improving.

No fine tuning is attempted at this stage.

In Appendix – F and Appendix – G you may find some part of the code that produced the results. The code was created with the Python programming.

The results are demonstrated in paragraph 4.1.

[12] C is the margin violation hyperparameter. For more details consult chapter 2.1.1

Testing Different DL architectures

After assessing the results (weighted F-score), different DL architectures will be tested with DL NN models with the use of TensorFlow and Keras. TensorFlow is used for Deep Learning because its speed is faster compared to the sklearn library; TensorFlow can exploit computer's GPU and with the use of the right software drivers and programming libraries, DL models' fitting speed may be up to x5 times faster.

Finding the optimum combination of Deep Learning Neural Network's hyperparameters, such as the number of hidden layers, the number of neurons per hidden layer and the batch size can be tough. Heaton (2008, pp. 128-129) provided some "rules" on that. Nevertheless, it is argued that there are no rules carved in stone on the approach; one may follow different paths until an optimum architecture is found (Geron, 2019, pp. 320-327). In this dissertation the following approaches will be implemented:

- Pyramid approach, where the number of the hidden units per layer decreases in each consecutive hidden layer (Geron, 2019, p. 325). For the purpose of simplification, the pyramid approach will be symbolized as ">"
- Reverse pyramid approach, where the number of the hidden units per layer increase in each consecutive hidden layer. For the purpose of simplification, the reverse pyramid approach will be symbolized as "<"
- A combination of pyramid and reverse pyramid architectures. More specifically, four different combinations will be tested:
    - A combination of a pyramid and reverse pyramid approach, which will be symbolized as "><"
    - A combination of reverse pyramid and pyramid approach, which will be symbolized as "<>"
    - Two consecutive pyramid architectures, symbolized ">>"
    - Two consecutive reverse pyramid architectures, symbolized "<<"
- Linear, where the number of hidden units is stable for each hidden layer (Geron, 2019, p. 325). This architecture will be symbolized as "|||"
- One hidden layer approach (Geron, 2019, p. 323), with symbol "|"

All of these approaches will be tested by using the "Sequential" method of the Keras API and a "Dense connection" between all hidden layers. The input layer's size will be equal to the best input layer size used in Classical ML tests. Also, on every test and on every hidden layer the "selu" activation function will be used and a dropout layer in between of all hidden layers will be added, with 30% chance of activating it. Moreover, in all architectures the "sigmoid" activation function will be added in the end, which will be considered as the output layer. The model will be compiled with the "Adam" optimizer and the "binary cross-entropy" loss function. Lastly, the model will be tested by using a validation split of 10%, a batch size of 32, shuffle, early stopping[13], and by integrating the dataset's class weights.

Because of using as input, a one-hot encoding input layer that was created with feature selection metrics, with a specific size, which does not represent sequence, DL NN cannot be implemented with alternative connection practices, such as LSTM[14]. LSTM needs data that is in sequence, thus much more data and time to be trained and in general a different approach in data preparation. (Vajjala, Majumder, Gupta, & Surana, 2020, p. 144).

The feature selection metric that will perform better during the experiments with the Classical ML models will be further used in Deep Learning.

Pyramid >

More specifically, in the pyramid approach, a minimum of 2 and a maximum of 15 hidden layers will be tested. So, eventually the model with the pyramid architecture will be tested 14 times (e.g., test1: 2 layers, test2: 3 layers, test3: 4 layers, etc.); for each consecutive layer, the number of hidden units will be reduced by 5%. In Figure 13 you may find an example of this architecture.

[13] Maximum epochs are equal to 100 and patience equal to 4. The validation loss will be monitored to refer if the model should not be further trained.

[14] Long-Short Term Memory

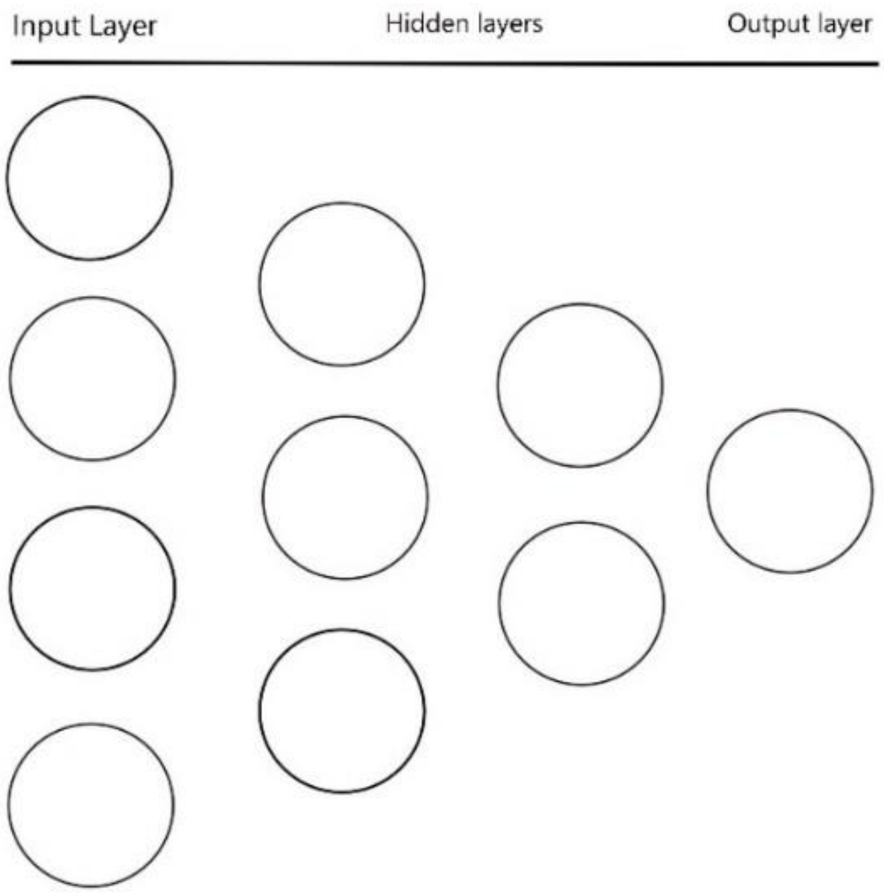


Figure 13: Example of pyramid architecture in a DL NN

Reverse pyramid <

In the reverse pyramid approach, a minimum of 2 and a maximum of 15 layers will be tested. So, eventually the model with the reverse pyramid architecture will be tested 14 times (e.g., test1: 2 layers, test2: 3 layers, test3: 4 layers, etc.); for each consecutive layer, the number of hidden units will be increased by 5%. In Figure 14 you may see an example of this architecture.

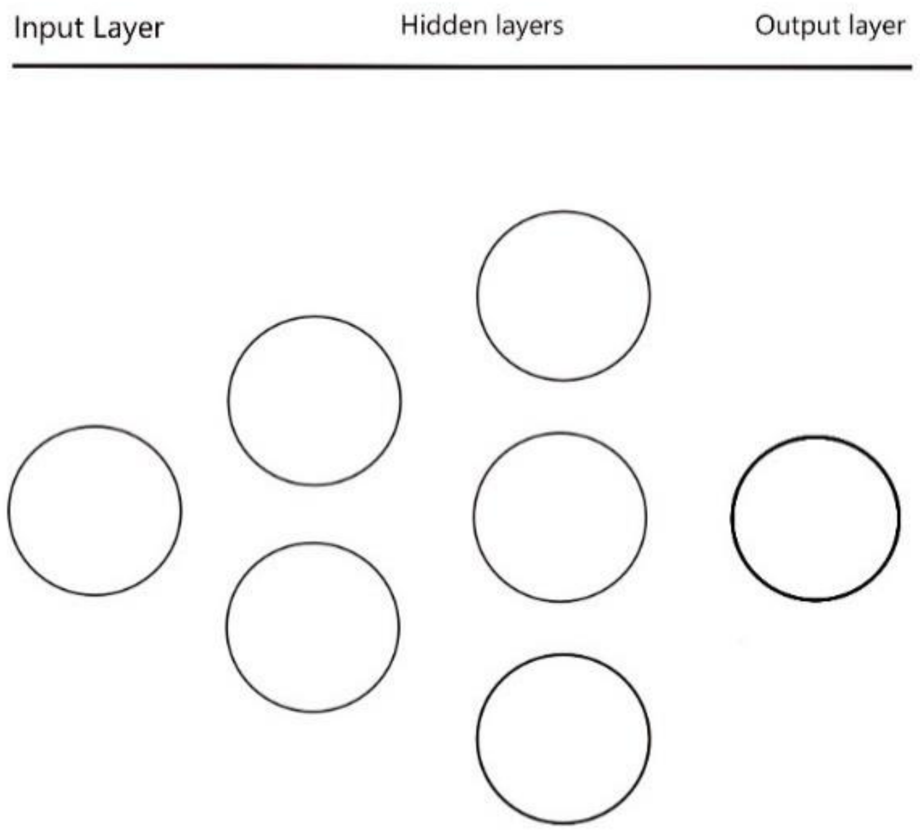


Figure 14: Example of reverse pyramid architecture in a DL NN

Combination of pyramid and reverse pyramid approach >< <> >> <<

Combining the pyramid and the reverse pyramid approach can result into many different layer architectures. Nevertheless, one could distinguish 4 main approaches. The first one starts with the pyramid approach and continues with the reverse pyramid architecture (e.g., Figure 15). The second one starts with the reverse pyramid approach, and it continues with the pyramid approach. The third one consists of two consecutive pyramid architectures and the last one of two consecutive reverse pyramid architectures. A total length of hidden layers amounting from 4 to 30 with a step of 2 will be tested (2 to 15, for each pyramid architecture). In each layer the increase/decrease of the hidden units will be at 5% of the previous layer's hidden units.

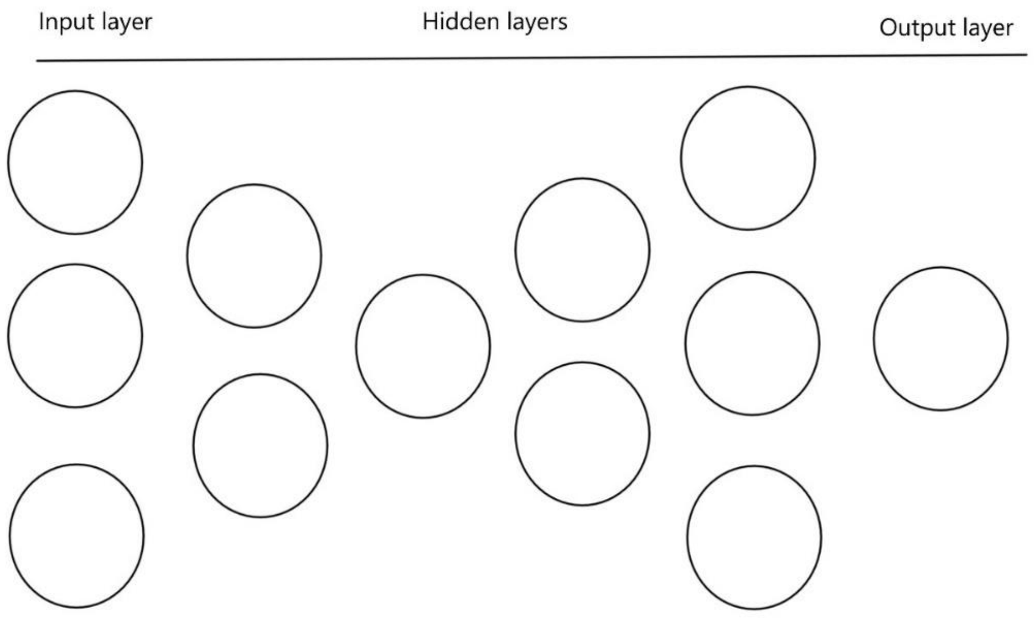


Figure 15: Example of combination of pyramid architecture in DL NN

Linear |||

Among the tested architectures is also the “linear” one. In this approach, the number of the hidden units will be the same along the length of the Neural Network (Figure 16). The result of this approach has no major difference compared to the pyramid approach according to Geron (Geron, 2019, p. 325) and sometimes it even performs better. A series of 2 to 15 layers will be added with a stable number of hidden layers each time. The number of the hidden layers will be stable during these tests and equal to 300 (as the length of the input data).

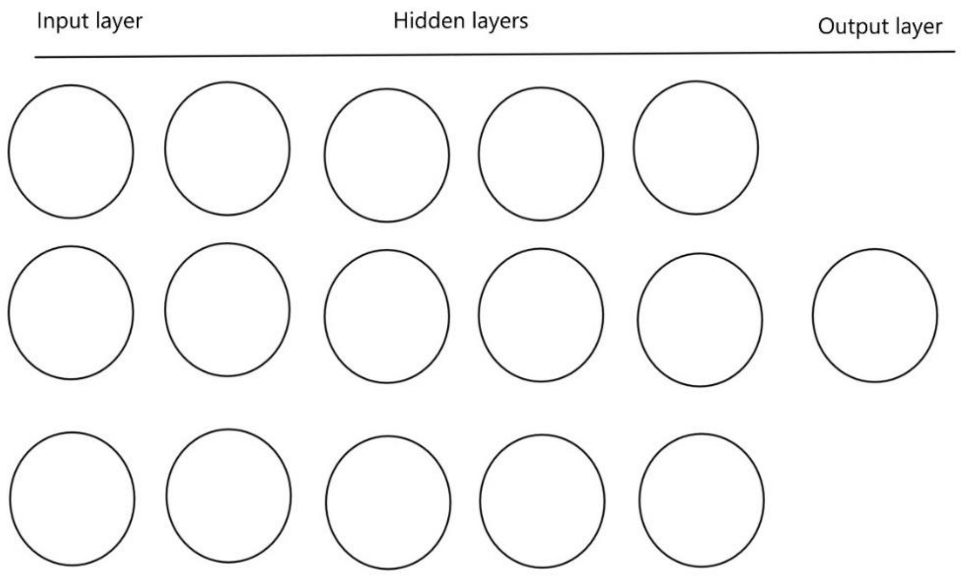


Figure 16: Example of a "linear" architecture in DL NN

One hidden layer approach |

Though a typical Neural Network consists of more than one hidden layer, one may start finding the ideal architecture by creating a one hidden layer Neural Network of Deep Learning. In some cases, the one hidden layer approach preforms very well (Geron, 2019, p. 323). For each feature selection metric, the one hidden layer will be tested.

Further testing DL Hyperparameters

After performing the above mentioned tests of DL NN, the architectures with the best results, will be further tested by implementing different hyperparameters, such as:

- Different number of hidden layers' units (e.g. double, triple, quadruple etc. of the size of the input layer)
- Different chances of dropout layer (from 10% chance to 90% chance)
- Different activation functions, namely "relu", "selu", "sigmoid", "softplus", "softsign", "tanh", "elu" and "exponential", which are callable via TensorFlow's Keras API.[15]

Last tests

Finally, further simple tests will be performed to compare them with the previous results and to have a general view on their potential. More particularly:

- As said in page 27, it is argued that DL doesn't need feature extraction, because DL NN can recognize which features are important or not to the model. Thus, five tests will be performed with all dataset's grouped features, by using a simple DL NN architecture that achieved the best F-score in previous tests with Deep Learning.
- A simple CNN and a simple LSTM DL NN architecture will be tested, five times.

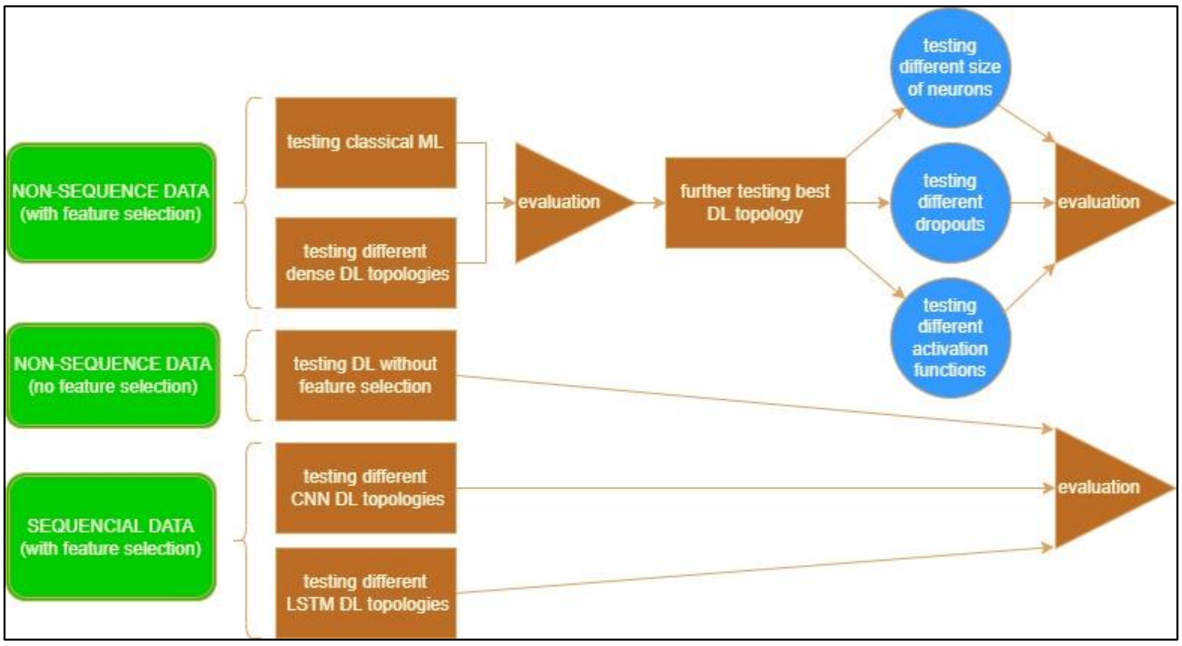
Final Remark

In Appendix – H You may find some piece of the code created with the Python programming language that runs the DL models, namely a pyramid architecture-topology, with the Keras programming library.

[15] Advise https://keras.io/api/layers/activations/ for further details on these functions.

Overview of Methodology

Given all the above, an overview of the methodology is depicted in Figure 17. You may notice that the focus is more on the classical Machine learning and the Deep Learning models with dense connections. Further analyzing all possible Deep Learning topologies would over-extend the length of the dissertation. Thus, it was decided to keep this approach and to analyze more in-depth the rest of the Deep Learning methods in a future work.



NON-SEQUENCE DATA (with feature selection)
NON-SEQUENCE DATA (no feature selection)
SEQUENCIAL DATA (with feature selection)
testing classical ML
testing different dense DL topologies
testing DL without feature selection
testing different CNN DL topologies
testing different LSTM DL topologies
evaluation
further testing best DL topology
testing different size of neurons
testing different dropouts
testing different activation functions
evaluation
evaluation

Figure 17: Overview of methodology

Chapter 4. Findings

As said in chapter 2, the Machine Learning models were initially trained and validated with simple or default parameters, via the sklearn library by using the Python programming language. Later, the dataset is tested with different NN Deep Learning models, with the use of TensorFlow and the Keras API.

4.1 Testing classical ML and DL models

The findings are demonstrated per Classifier. The results of Precision, Recall and F_1-score are the weighted results, which means that the distribution of the classes was taken into account to calculate them.

In matrices from 5 to 10 you may find the results of the Naïve Bayes, k-Nearest Neighbours, Support Vector Machine, Random Forest, Logistic Regression and Deep Learning Classifiers.

As you can see in Table 5, the chi square (x^2) metric "produces" the highest weighted F_1-score (72,8%) compared to devmax.DF and tf.idf.

Table 5: Results of testing Naïve Bayes with different feature selection metrics and different input vector size (words)

metric	Vector Size	wPrecision	wRecall	wFscore
x^2	100	83,2%	62,0%	70,3%
x^2	200	76,6%	69,9%	72,8%
x^2	300	73,6%	71,4%	72,2%
x^2	400	67,0%	71,8%	69,0%
tf-idf	100	53,0%	30,9%	38,3%
tf-idf	200	66,1%	53,8%	59,1%
tf-idf	300	63,2%	60,7%	61,7%
tf-idf	400	61,5%	65,5%	63,1%
devmax	100	81,0%	64,8%	71,1%
devmax	200	70,4%	70,0%	69,6%
devmax	300	67,8%	72,6%	69,7%
devmax	400	66,1%	72,4%	68,7%

In Table 6 you can notice that by using the F_1-score metric, devmax.DF outperformed chi square and tf.idf. Nevertheless, it seems that the k-Nearest Neighbors model, with k=1 does not perform as well as the Naïve Bayes model.

Table 6: Results of testing kNN with different feature selection metrics and different input vector size (words)

metric	Vector Size	wPrecision	wRecall	wFscore
x^2	100	54,8%	23,8%	32,8%
x^2	200	65,7%	18,8%	28,6%
x^2	300	71,0%	16,9%	26,3%
x^2	400	71,8%	16,1%	25,2%
tf-idf	100	30,8%	17,6%	21,9%
tf-idf	200	46,7%	17,7%	25,2%
tf-idf	300	53,4%	17,4%	25,5%
tf-idf	400	59,7%	16,3%	24,6%
devmax	100	57,6%	52,9%	52,5%
devmax	200	60,5%	40,2%	46,4%
devmax	300	77,0%	34,9%	47,6%
devmax	400	77,5%	31,4%	44,2%

In Table 7 you can notice that by using the F_1-score metric, chi square outperformed devmax.DF and tf.idf.

Table 7: Results of testing SVM with different feature selection metrics and different input vector size (words)

metric	Vector Size	wPrecision	wRecall	wFscore
x^2	100	86,7%	69,9%	75,5%
x^2	200	85,2%	79,5%	81,7%
x^2	300	74,4%	78,3%	76,0%
x^2	400	69,7%	77,5%	73,1%
tf-idf	100	49,0%	33,9%	39,5%
tf-idf	200	71,2%	61,1%	64,7%
tf-idf	300	70,1%	66,2%	67,4%
tf-idf	400	64,9%	70,5%	67,1%
devmax	100	83,9%	75,6%	79,1%
devmax	200	80,9%	79,6%	79,9%
devmax	300	76,2%	80,1%	77,9%
devmax	400	72,9%	79,5%	75,9%

In Table 8, it seems that devmax.DF outperformed the other metrics. Moreover, it is noticed that this Random Forest model reached high precision scores.

Table 8: Results of testing Random Forest with different feature selection metrics and different input vector size (words)

metric	**Vector Size**	**wPrecision**	**wRecall**	**wFscore**
x^2	100	90,2%	63,3%	72,7%
x^2	200	91,9%	64,8%	75,2%
x^2	300	93,9%	54,8%	67,9%
x^2	400	94,2%	50,3%	64,1%
tf-idf	100	52,2%	30,5%	37,7%
tf-idf	200	83,9%	51,9%	62,2%
tf-idf	300	90,3%	47,4%	59,9%
tf-idf	400	92,8%	43,8%	57,4%
devmax	100	89,5%	70,8%	78,3%
devmax	200	91,9%	67,1%	76,8%
devmax	300	93,5%	59,2%	71,5%
devmax	400	94,5%	51,1%	65,0%

In Table 9, devmax.DF outperformed the other metrics. Moreover, it seems that Logistic Regression reached high weighted F-scores, compared to the previous Machine Learning models.

Table 9: Results of testing Logistic Regression with different feature selection metrics and different input vector size (words)

metric	**Vector Size**	**wPrecision**	**wRecall**	**wFscore**
x^2	100	85,8%	67,9%	74,5%
x^2	200	87,3%	76,2%	80,9%
x^2	300	86,7%	76,9%	81,3%
x^2	400	87,0%	76,6%	81,2%
tf-idf	100	54,5%	33,6%	40,1%
tf-idf	200	74,7%	58,4%	64,8%
tf-idf	300	80,2%	65,0%	71,2%
tf-idf	400	83,1%	70,6%	76,0%
devmax	100	87,2%	73,2%	79,1%
devmax	200	87,4%	76,9%	81,6%
devmax	300	87,4%	78,6%	82,6%
devmax	400	87,8%	78,3%	82,5%

Lastly, in Table 10 you can notice that devmax.DF reached better results (regarding the weighted F_1-score metric), compared to the other metrics. It is reminded, that this Deep Learning model is shallow and there was no effort to make it more complex at this stage. Later, a Deep Learning model will be trained and tested again, with different and more complex hyperparameters.

Table 10: Results of testing Deep Learning (simple layer) with different feature selection metrics and different input vector size (words)

metric	Vector Size	wPrecision	wRecall	wFscore
x^2	100	84,5%	67,2%	73,5%
x^2	200	85,3%	76,1%	80,1%
x^2	300	85,3%	75,1%	79,5%
x^2	400	84,5%	75,2%	79,3%
tf-idf	100	54,8%	33,4%	39,3%
tf-idf	200	73,4%	56,9%	63,1%
tf-idf	300	77,0%	61,6%	67,5%
tf-idf	400	79,8%	67,9%	72,8%
devmax	100	85,4%	72,5%	78,0%
devmax	200	84,6%	76,0%	79,9%
devmax	300	84,1%	78,7%	81,1%
devmax	400	84,5%	76,9%	80,2%

In Table 11 you may find an overview of the highest weighted F-score per classifier from Table 5 to Table 10.

Table 11: Best weighted F-scores of running Classical ML models and DL via the sci-kit Python Library with different parameters

Classifier	metric	Vector Size	Precision	Recall	F-score
NB	x^2	200	76,6%	69,9%	72,8%
k-NN	devmax	100	57,6%	52,9%	52,5%
SVM	x^2	200	85,2%	79,5%	81,7%
RF	devmax	100	89,5%	70,8%	78,3%
LR	devmax	300	87,4%	78,6%	82,6%
DL	devmax	300	84,1%	78,7%	81,1%

Conclusions:

- The highest weighted F_1-score (82,6%) is with the devmax.DF feature selection metric, with a Logistic Regression ML model and a Lexicon of 300 terms.
- The metric Devmax.DF achieved higher F-score with kNN, RF, LR, and DL models, whereas the x^2 metric achieved higher F-score with NB and SVM.
- Performance in DL (Table 10) between devmax.DF and x^2 is relatively close (1%).

4.2 Testing different DL NN architectures

In this paragraph you may find the results of running different DL NN architectures. Only the feature selection metrics x^2 and devmax.DF are further tested because they performed better than tf.idf according to the results of paragraph 4.1. Moreover, it seems that x^2 and devmax.DF had a close match (Table 10 & Table 11). Thus, it is reasonable to use them both in the initial stage of the further tests.

It is reminded that:

- A Sequential model of Keras API is used with Dense connection between the hidden layers.
- The "Selu" activation function is used.
- A dropout layer is added between all hidden layers, with a dropout chance of 30%.
- The input layer's length is equal to 300, because this input layer length performed better compared to other lengths in the initial test of DL (Table 10).
- The output layer uses the "sigmoid" activation function.
- The model is compiled with the "Adam" optimizer (learning rate =0.0001) and the "binary crossentropy" loss function.
- Early stopping is activated, which prevents further training when cross validation loss is not improving; a batch size of 32 was used; shuffle was enabled (which automatically shuffles the data with each training cross validation).

The architectures to be tested are split into 9 categories:

1. "Pyramid approach" >
2. "Reverse pyramid approach" <
3. Combination of "pyramid and reverse pyramid approach" ><
4. Combination of "reverse pyramid and pyramid approach" <>
5. Combination of "two consecutive pyramid approaches" >>
6. Combination of "two consecutive reverse pyramid approaches" <<
7. "Linear approach" |||
8. "One-layer approach" |

Chi square (x^2)

After running 99 times different DL architectures, you may find in Table 12 the best F-scores for each DL architecture that was tested. You can find the whole matrix in the Appendix - A.

Table 12: Best F-scores per DL architecture by using the x^2 feature extraction metric

Architecture	Hidden layers	wPrecision	wRecall	wFscore
reverse pyramid <	2	85,9%	80,9%	82,2%
pyramid >	2	85,9%	80,9%	82,2%
combination of pyramids ><	4	86,0%	79,3%	80,9%
combination of pyramids <>	4	85,9%	79,6%	81,2%
combination of pyramids >>	4	86,4%	79,6%	81,3%
combination of pyramids <<	4	86,2%	80,0%	81,5%
linear metric |||	3	86,1%	80,7%	82,1%
one layer approach metric |	1	86,0%	81,9%	82,9%

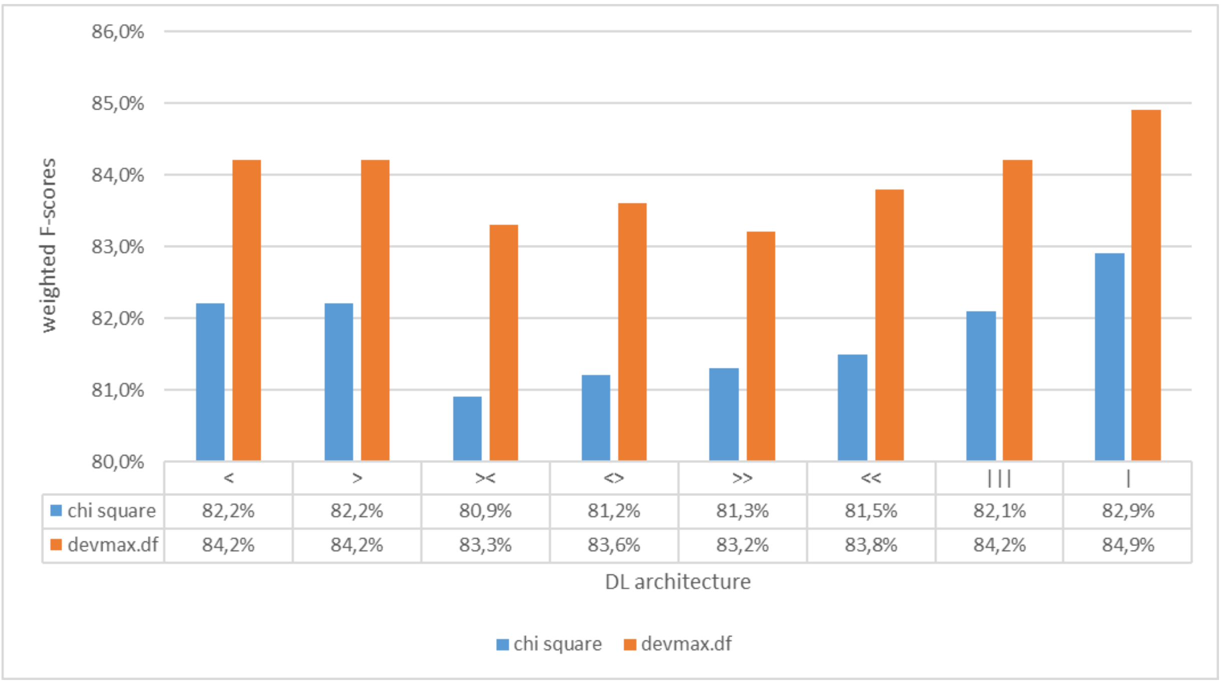
Devmax.DF

After running 99 times different DL architectures, you may find in Table 13 the best F-scores for each DL architecture that was tested. You can find the whole matrix in the Appendix - A.

Table 13: Best F-scores per DL architecture by using devmax.DF feature extraction metric

Architecture	Hidden layers	wPrecision	wRecall	wFscore
pyramid >	2	85,7%	84,0%	84,2%
reverse pyramid <	2	85,6%	84,0%	84,2%
combination of pyramids ><	4	86,1%	82,6%	83,3%
combination of pyramids <>	4	85,9%	82,9%	83,6%
combination of pyramids >>	4	85,5%	82,8%	83,2%
combination of pyramids <<	4	85,7%	83,4%	83,8%
linear metric |||	3	86,1%	83,9%	84,2%
one layer approach metric |	1	86,2%	84,6%	84,9%

In Figure 18 you can see the weighted F-scores of chi square and devmax.DF as shown in Tables 12 and 13. You can notice in Figure 18 that devmax.DF outperformed chi square. Moreover, in Figure 19 there is a 3D demonstration of the weighted F-score results as shown in Tables 12 and 13.



	<	>	><	<>	>>	<<	|||	|
🟦 chi square	82,2%	82,2%	80,9%	81,2%	81,3%	81,5%	82,1%	82,9%
🟧 devmax.df	84,2%	84,2%	83,3%	83,6%	83,2%	83,8%	84,2%	84,9%

Figure 18: Best F-scores per DL architecture by using chi square and devmax.DF feature extraction metric

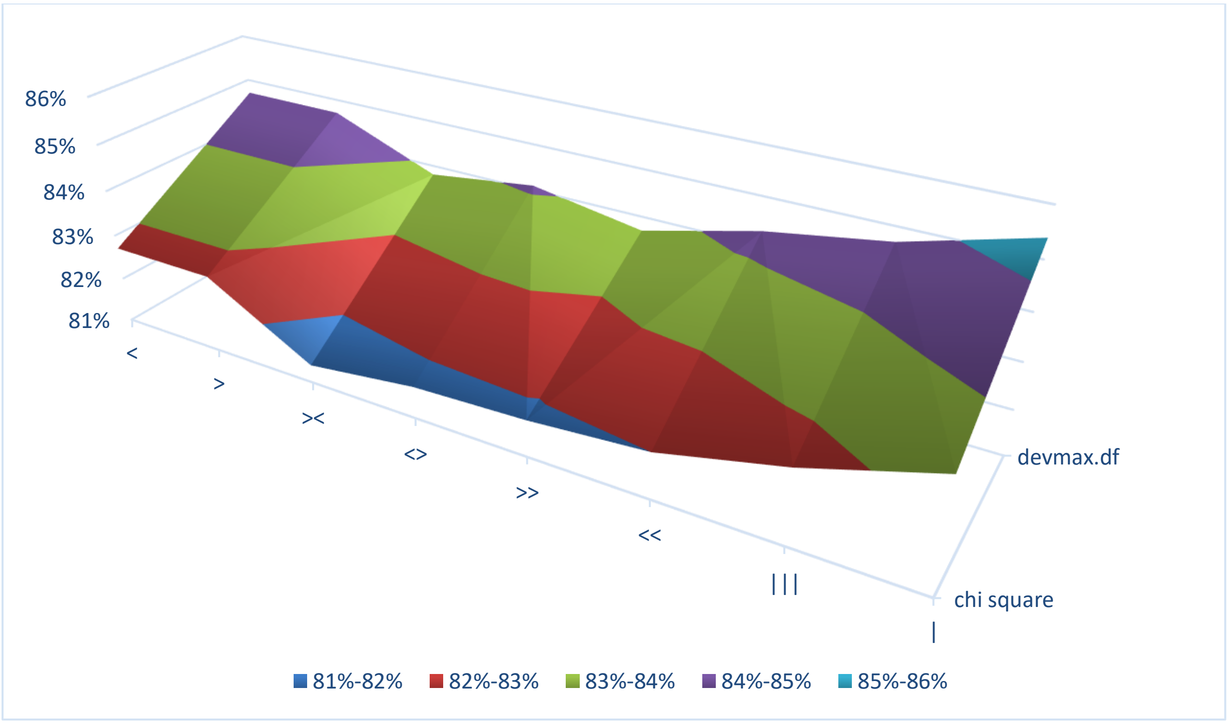


Figure 19: 3D demonstration of best F-scores per DL architecture by using chi square and devmax.DF feature extraction metric

Conclusions:

- The one-layer architecture performed better than the other architectures (Figure 17).
- The metric Devmax.DF performed better than x^2 (Figure 17).
- DL architectures with fewer hidden layers performed better than DL architectures with more hidden layers.

Remark:

Since the feature extraction metric devmax.DF performed better than the x^2 metric, hereafter only the devmax.DF metric will be used for further testing.

Further testing the one-layer approach

In Table 13, it is shown that a DL NN with only one hidden layer performed the best F-score. To further examine the capabilities of the one-layer approach, it will be tested with different volumes of units (of the one hidden layer). Particularly, the tests will be performed by increasing in each test the hidden layer's units by 2, 3, 4, …, and up to 200 times of the length of the input value (=300). In Figure 20 you can see an overview of the results, while in the Appendix – B you can find the whole dataset.

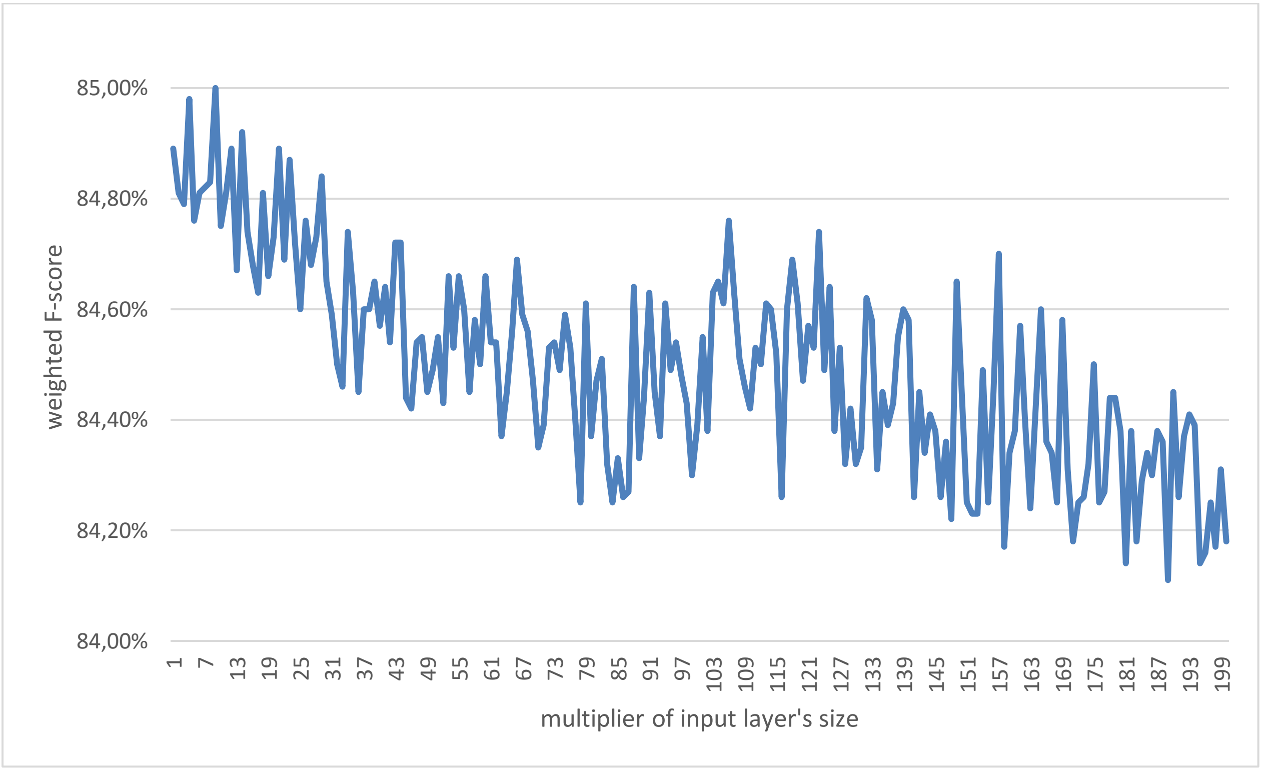


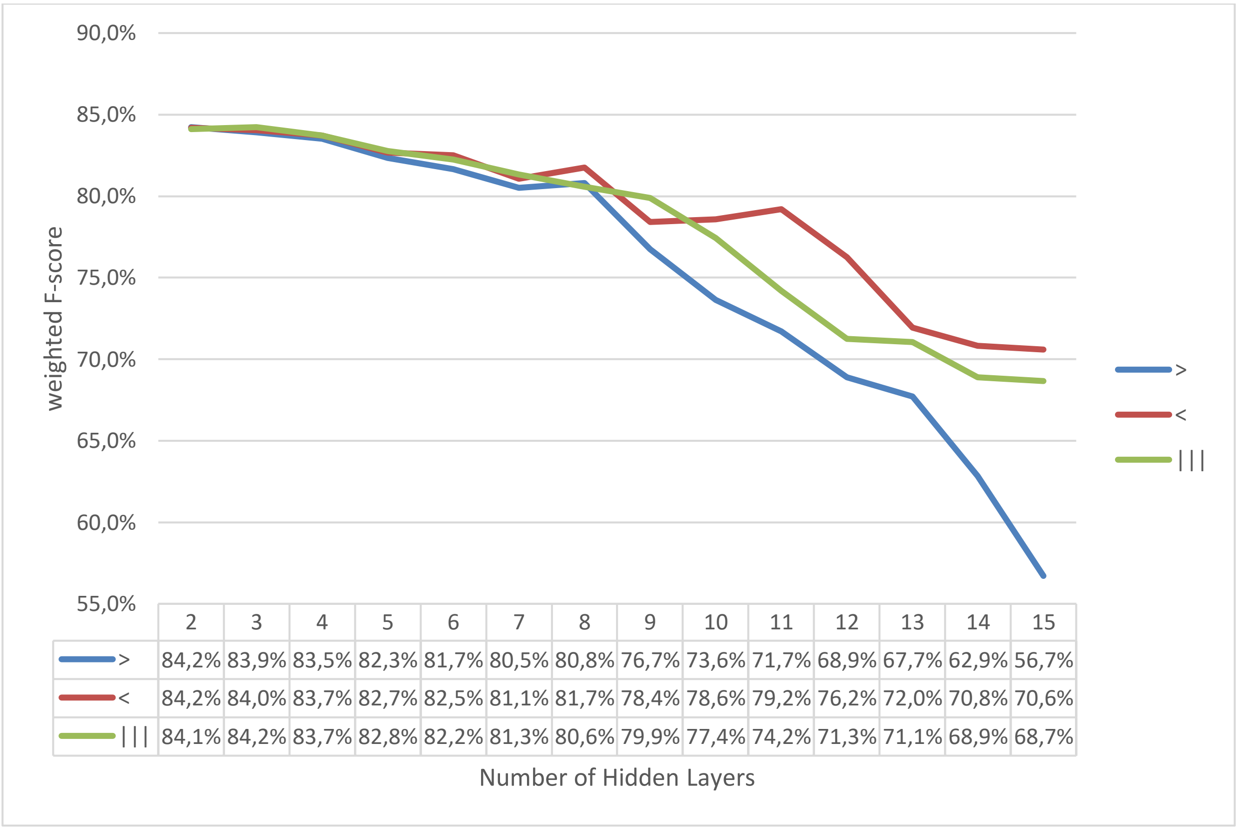
Figure 20: Weighted F-scores, by using only one hidden layer, and by increasing the hidden layer's units with a multiplier

Conclusions:

- The weighted F-scores span between 84.11% and 85.00%.
- Though the range between maximum and minimum weighted F-score is insignificant, the top weighted 10 F-scores were achieved with a smaller multiplier than a higher one.
- The highest weighted F-score is achieved with a multiplier of 9, which resulted a weighted F-score of 85%.

Further testing the pyramid approaches and the linear approach

In Table 13, you can notice that the second best F-score was achieved with the pyramid, reverse pyramid, and linear architecture (all of them reached a weighed 84.2% F-score). A further analysis of the data of the table in the Appendix - A, shows that the reverse pyramid and the linear architecture is more persistence to F-score drop while the number of the hidden layers increases (Figure 21).



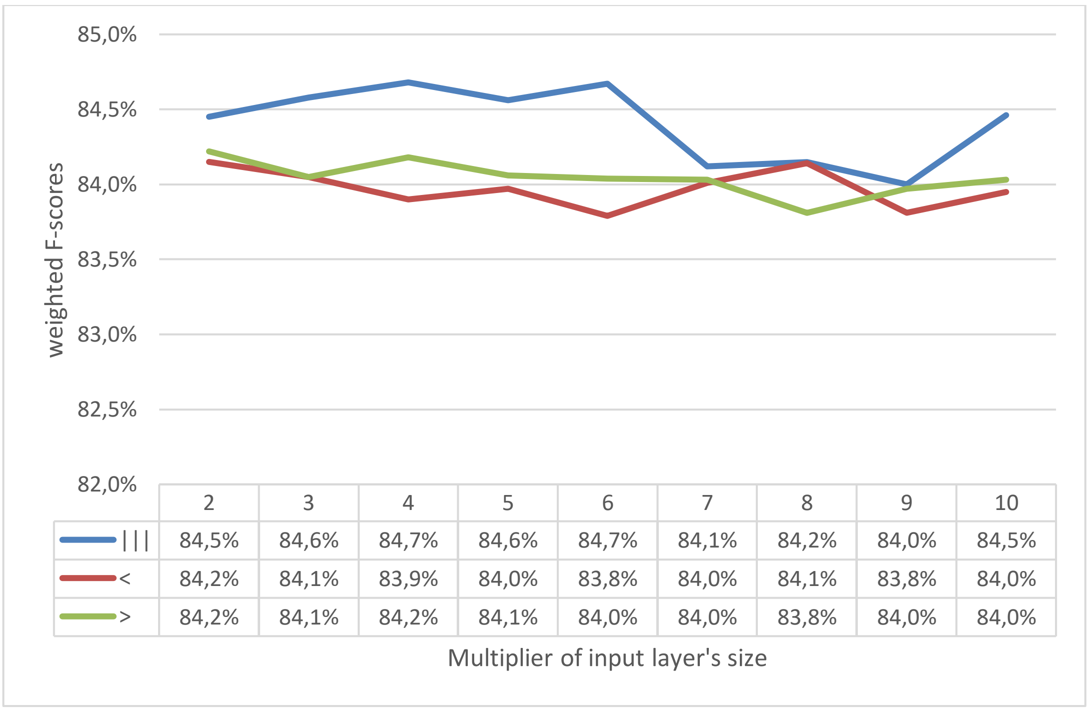
	2	3	4	5	6	7	8	9	10	11	12	13	14	15
>	84,2%	83,9%	83,5%	82,3%	81,7%	80,5%	80,8%	76,7%	73,6%	71,7%	68,9%	67,7%	62,9%	56,7%
<	84,2%	84,0%	83,7%	82,7%	82,5%	81,1%	81,7%	78,4%	78,6%	79,2%	76,2%	72,0%	70,8%	70,6%
|||	84,1%	84,2%	83,7%	82,8%	82,2%	81,3%	80,6%	79,9%	77,4%	74,2%	71,3%	71,1%	68,9%	68,7%

Figure 21: Weighted F-scores of 3 NN DL architectures, by increasing the number of the hidden layers, and by using the devmax.DF feature selection metric

Given the above, the pyramid, reverse pyramid and linear approach will be further tested. All test were performed with a two-hidden layer NN DL model. More particularly:

- The pyramid architecture is tested with different sizes of the first hidden layer’s units. Particularly, the first layer will be multiplied by 2, 3, 4, …, 10 times more than the size of the input layer’s size. The second layer’s units will remain equal to the input layer’s size.
- The reverse pyramid is tested by increasing the second hidden layer’s units, by 2, 3, 4, …, 10 times. The first hidden layer’s units will remain equal to the input layer’s size.
- The linear approach architecture is tested by increasing all the hidden layers’ units by 2, 3, 4, …, 10 times more than the size of the first hidden layer’s units.

In Figure 22 you can see the results.



	2	3	4	5	6	7	8	9	10
🟦 |||	84,5%	84,6%	84,7%	84,6%	84,7%	84,1%	84,2%	84,0%	84,5%
🟥 <	84,2%	84,1%	83,9%	84,0%	83,8%	84,0%	84,1%	83,8%	84,0%
🟩 >	84,2%	84,1%	84,2%	84,1%	84,0%	84,0%	83,8%	84,0%	84,0%

Figure 22: Weighted F-scores of the pyramid, reverse pyramid, and linear architectures with different sizes of hidden units, by using the devmax.DF feature selection metric

Conclusions:

- It seems that there is no tendency that can lead to higher weighted F-scores.
- For multipliers 2, 3 and 4, the weighted F-scores are very close, with all DL NN architectures.
- Performance of reverse pyramid “<”, is more stable while the multiplier increases, compared to the other architectures, which perform worse while the multiplier increases.
- The one-layer architecture performed better (85.00% weighted F-score) than the pyramid, reverse pyramid, and linear architecture.

Remark:

Since the one-hidden layer architecture performed better, hereafter only this architecture will be further tested.

Dropout & multiplier with one-hidden layer DL NN

Further tests were performed with the one-hidden layer architecture. Particularly, this schema was tested with different hidden units’ size (multiplied by 1, 2, 3, …, 50 of the input size) and different dropout chance (from 10% to 90%) of the hidden layer’s units (Table 14).

Table 14: Weighted F-score performance of a one-hidden layer DL NN, testing it with different hidden units' size and different dropout chance

	Dropout Chance								
Multiplier of input layer size	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**
1	84,7%	85,0%	84,7%	84,6%	84,7%	84,5%	84,1%	83,2%	76,1%
2	84,7%	85,0%	84,8%	84,8%	84,8%	84,7%	84,7%	84,2%	82,8%
3	85,0%	84,8%	84,8%	84,9%	84,7%	**85,1%**	84,8%	84,6%	83,8%
4	84,8%	84,9%	84,9%	84,9%	84,7%	85,0%	85,0%	84,6%	84,4%
5	84,7%	84,8%	84,8%	84,9%	85,0%	84,9%	85,0%	84,9%	84,3%
6	84,9%	84,9%	84,9%	84,9%	85,0%	84,8%	84,8%	84,9%	84,6%
7	84,8%	84,8%	84,9%	84,9%	85,0%	84,9%	85,0%	85,0%	84,7%
8	84,8%	84,8%	85,0%	85,0%	85,0%	85,0%	84,8%	84,8%	84,6%
9	84,8%	84,9%	85,0%	84,9%	84,8%	84,8%	84,8%	84,9%	84,8%
10	84,9%	84,9%	85,0%	85,0%	84,9%	84,9%	84,9%	85,0%	84,8%
11	84,8%	85,0%	84,9%	84,9%	84,8%	84,9%	84,9%	85,0%	84,7%
12	84,8%	84,9%	84,9%	84,9%	84,9%	84,9%	84,9%	85,0%	84,9%
13	84,9%	84,9%	84,8%	84,9%	84,8%	85,0%	84,9%	85,1%	84,8%
14	84,8%	84,8%	84,9%	84,8%	84,9%	84,9%	85,0%	84,9%	84,9%
15	84,8%	84,9%	84,8%	84,8%	84,8%	85,0%	84,9%	85,0%	85,0%
16	84,9%	84,8%	84,8%	84,8%	85,0%	85,1%	85,0%	85,0%	84,8%
17	84,8%	84,8%	84,8%	84,8%	84,9%	84,9%	85,1%	84,9%	84,9%
18	84,8%	84,9%	84,8%	84,9%	84,9%	85,0%	84,8%	84,9%	84,7%
19	84,8%	84,9%	84,8%	84,9%	85,0%	84,9%	84,9%	85,0%	84,9%
20	84,8%	84,8%	84,9%	84,8%	84,8%	85,0%	85,0%	84,9%	84,9%
21	84,8%	84,8%	84,8%	84,9%	85,0%	84,8%	84,9%	85,0%	84,9%
22	84,6%	84,8%	84,9%	84,8%	84,9%	84,9%	84,9%	85,1%	85,0%
23	84,9%	84,9%	84,9%	85,0%	84,8%	84,9%	85,0%	84,7%	85,0%
24	85,0%	84,7%	84,8%	84,8%	84,8%	84,8%	84,9%	84,9%	84,9%
25	84,9%	84,7%	84,7%	85,0%	84,9%	85,0%	85,0%	84,8%	84,9%
26	84,8%	84,8%	84,8%	84,9%	84,8%	85,1%	84,9%	84,9%	85,0%
27	84,9%	84,8%	84,9%	84,8%	84,8%	85,0%	84,9%	85,0%	84,9%
28	84,8%	84,7%	84,7%	85,0%	84,9%	84,8%	84,9%	85,0%	84,9%
29	84,8%	84,8%	84,9%	84,8%	84,7%	84,8%	85,0%	84,9%	85,0%
30	84,7%	84,8%	84,7%	84,9%	84,8%	84,8%	84,9%	85,0%	85,1%
31	84,7%	84,9%	84,8%	84,7%	84,9%	84,9%	85,0%	85,0%	85,0%
32	84,7%	84,8%	84,9%	84,8%	84,8%	84,8%	84,9%	84,8%	85,1%
33	84,7%	84,9%	84,8%	84,8%	84,8%	84,9%	84,9%	85,0%	85,1%
34	84,6%	84,7%	84,9%	84,7%	84,8%	84,9%	85,0%	84,8%	85,0%
35	84,7%	84,7%	84,7%	84,7%	84,8%	85,0%	85,0%	84,9%	85,0%
36	84,5%	84,8%	84,7%	84,9%	84,7%	84,8%	85,0%	84,8%	85,0%
37	84,7%	84,9%	84,6%	84,8%	84,9%	84,8%	85,0%	85,1%	84,9%
38	84,6%	84,9%	84,6%	84,8%	84,7%	84,7%	84,9%	85,0%	85,0%
39	84,8%	84,7%	84,9%	84,8%	84,9%	84,9%	84,9%	84,8%	85,0%
40	84,6%	84,8%	84,8%	84,8%	84,7%	84,8%	84,9%	85,0%	84,9%
41	84,6%	84,7%	84,8%	84,9%	84,8%	84,7%	84,9%	84,9%	85,1%
42	84,7%	84,5%	84,6%	84,9%	84,9%	84,8%	85,0%	84,9%	85,1%
43	84,7%	84,7%	84,8%	84,8%	84,8%	84,8%	85,0%	85,0%	85,1%
44	84,6%	84,7%	84,8%	84,7%	84,8%	84,7%	85,0%	84,9%	85,1%
45	84,6%	84,5%	84,8%	84,9%	84,8%	84,9%	84,8%	85,0%	**85,2%**
46	84,8%	84,6%	84,6%	84,8%	84,9%	84,9%	84,9%	84,9%	85,0%
47	84,7%	84,8%	84,6%	84,8%	84,9%	84,7%	84,9%	85,0%	85,1%
48	84,6%	84,4%	84,9%	84,8%	84,7%	84,9%	84,8%	85,1%	85,0%
49	84,7%	84,5%	84,8%	84,6%	85,0%	84,8%	84,8%	85,0%	85,1%
50	84,5%	84,7%	84,5%	84,5%	84,8%	84,6%	84,7%	84,6%	84,6%

Conclusions:

- The average weighted F-score of these tests is equal to 84.8%, spanning in a range between 76.1% and 85.2%.
- The highest weighted F-score was achieved with a hidden unit's length equal to the input layers size multiplied by 45 and a dropout chance of 90%.

Remarks:

- Though, the best weighted F-score was achieved with a 90% dropout chance and hidden units' size equal to the input layer size multiplied by 45, these parameters won't be used further. The 90% dropout chance in too high and the X45 multiplier will increase computational needs more computational power.
- Alternatively, hereafter a DL NN of one-hidden layer and hidden units' size equal to the input layer size multiplied by 3 and a dropout layer with 60% will be used. This schema performed relatively good as you can notice in Table 14.

Testing different Activation Functions

Furthermore, different activation functions were tested by using a one-hidden layer DL NN with hidden units equal to 3X(300)[16] and a dropout layer of 60% chance. More specifically, the available activation functions of Keras were used, namely "relu", "selu", "sigmoid", "softplus", "softsign", "tanh", "elu" and "exponential". You can see the results in Table 15. It seems, that "selu", "softsign", "tanh" and "elu" perform better than the other activation functions, for this specific dataset/task.

Table 15: Performance of testing different activation functions with a one-hidden layer DL NN

activation function	wPrecision	wRecall	wFscore
relu	87,1%	82,4%	83,9%
selu	87,0%	83,6%	84,5%
sigmoid	87,2%	71,8%	75,6%
softplus	87,7%	74,0%	77,5%
softsign	86,5%	83,7%	84,4%
tanh	86,3%	83,8%	84,4%
elu	86,2%	84,0%	84,5%
exponential	87,3%	74,2%	77,4%

[16] The input layer size is equal to 300.

Further tests

Further tests were performed to give a more round view on different DL NN architectures and methods, and to highlight their potential to the extend it is possible within this dissertation.

Firstly, five different simple tests with a simple DL NN of one-hidden layer and hidden units equal to 1 ,2, 3, 4, 5X(length of input layer) and a dropout layer of 60% chance are performed by using all dataset's grouped features, which are 5,559 (Table 16). These tests are performed because as said previously, it is argued that in DL one should not use feature selection, because DL NN can recognize important and ignore unimportant features.

Secondly, five simple CNN DL architectures were tested with an Embedding[17] layer and a Conv1D[18] layer (Table 17). To create the Embedding layer, the dataset's rows were transformed into vectors, by using a Lexicon size equal to 300 (and by using devmax.DF). This is the same Lexicon that was created and used in the previous tests. Each word of the sentence is represented with the Lexicon's index number. Those words from each sentence that were not found in the Lexicon of the 300 words were ignored and were not put into the sentence's vector. After transforming each row of data into a vector, it is noted that the minimum sentence length is 0, the maximum sentence length is equal to 51, the average 4,64 and the standard deviation of the dataset is equal to 4,48. Having the above in mind the Embedding layer's parameters were set as:

- input_dim = 300 (which corresponds to the Lexicon's length that was used in previous tests)
- output_dim = 51 (which is equal to the maximum(Sentence length of dataset))
- input_length = 51 (which is equal to the maximum(Sentence length of dataset))

Also, a CNN layer was added with parameters:

- Filter_Size = 300 (which corresponds to the Lexicon's length that was used)
- Kernel_size = 4 (groups 4 vectors in each sentence. Actually, captures statistics of 4 consecutive words)
- The "relu" activation functions is used.

Finally, a GlobalMaxPool1D[19] layer is added.

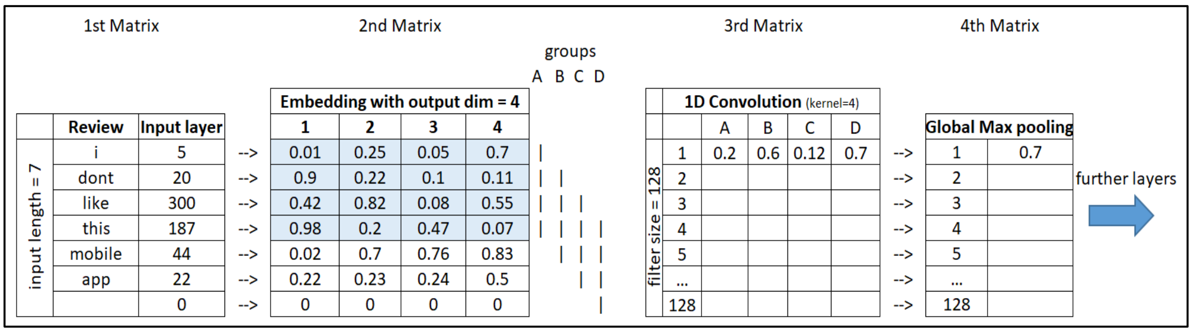
[17] https://keras.io/api/layers/core_layers/embedding/

[18] https://keras.io/api/layers/convolution_layers/convolution1d/

[19] https://keras.io/api/layers/pooling_layers/global_max_pooling1d/

In Figure 23, you can find an example of the topology related to the CNN model. As you can see, the input layer has a specific length (input length = 7). If there are less words in a review, like in the example of Figure 21, the rest of the layer contains zero values. Further, each Input layer of 1st Matrix is transformed into a 2-dimensional array via the Embedding layer (2nd Matrix). The new dimension is dependent on the “outpud_dim” parameter of the Embedding layer. In the example the output_dim is set equal to 4. Thus, in this demonstrated example the word “dont” became a four-length vector equal to [0.9, 0.22, 0.1, 0.11]. Similarly, the other word became a four-length vector. The whole Review (1st Matrix) eventually became a 2-dimensional array with dimensions 7 X 4 (where 7 = input_length and 4 = output_dim).

The 3rd Matrix of the example is the CNN layer with parameters filter_size=128 and kernel=4. The filter size as said above, is actually the layer’s output. The kernel defines the grouping of consecutive data from the Embedding layer (2nd Matrix). So, if kernel=4 then one can retrieve 4 groups of data (A, B, C, D). Group A is colored with a smooth blue color in the example (2nd Matrix). In each group, arbitrary weights and biases are applied via the CNN layer, and finally the output of each group is put in the 3rd Matrix in the way it is depicted. The filter size is equal to 128, so this procedure is repeated 128 times. Finally, the GlobalMaxPooling layer (4th Matrix), filters each row of the CNN layer (3rd Matrix) and outputs the maximum number of each row. After this procedure one may add any other hidden layer, such as a Dense layer, or even a second CNN layer right after the first CNN layer.



Review	Input layer
i	5
dont	20
like	300
this	187
mobile	44
app	22
	0

Embedding with output dim = 4			
1	2	3	4
0.01	0.25	0.05	0.7
0.9	0.22	0.1	0.11
0.42	0.82	0.08	0.55
0.98	0.2	0.47	0.07
0.02	0.7	0.76	0.83
0.22	0.23	0.24	0.5
0	0	0	0

1D Convolution (kernel=4)				
	A	B	C	D
1	0.2	0.6	0.12	0.7
2				
3				
4				
5				
...				
128				

Global Max pooling	
1	0.7
2	
3	
4	
5	
...	
128	

Figure 23: Demonstration example of a CNN layer

Tests of CNN are performed by using different filter sizes, namely multiplies of the Lexicon’s length and by keeping all other parameters stable. In Appendix – I you can find some of the code that was used to create the CNN model.

Finally, five simple LSTM DL architectures were tested with Embedding layer and LSTM layer (Table 18). The Embedding layer parameters are the same as those used in the CNN previously.

Furthermore, an LSTM layer is added with the following parameters:

- Units = 300
- Dropout = 0.2
- Activation function = “tanh”
- Return_sequence = True

Tests of LSTM are performed by using different unit sizes, namely multiplies of the Lexicon’s length and by keeping all other parameters stable.

Table 16: Weighted F-scores of testing a simple one hidden-Dense layer with all dataset’s vector size (grouped features/words 5,559) and with hidden layer’s size multiplied by 1, 2, 3, 4, 5 times.

Multiplier	wF-score
1	79,9%
2	80,0%
3	80,4%
4	80,1%
5	80,0%

Table 17: Weighted F-scores of testing a simple CNN layer, by changing the hidden layer’s “Filter size” (layer’s output) with each test

Multiplier	wF-score
1	83,5%
2	83,7%
3	83,8%
4	83,7%
5	83,5%

Table 18: Weighted F-scores of testing a simple LSTM layer, by changing the hidden layer’s “units” (layer’s output) with each test

Multiplier	wF-score
1	74,9%
2	77,4%
3	77,4%
4	78,1%
5	76,7%

<u>Remarks:</u>

- Results of Table 16 show that using all dataset’s features can provide relatively good results (around 80% weighted F-score), compared to the simple DL NN architecture that was tested, which produced 81,1% weighted F-score (Table 11). Further research with different hyperparameters and more hidden layers, may provide even better results, thus, further research is needed to confirm that.
- Results of Table 17 show that a simple CNN schema provided good results (83,8% weighted F-score). Further research with different hyperparameters and/or more hidden layers, may lead to even better performance.
- Results of Table 18 show that a simple LSTM does not perform well. Further research is needed with different hyperparameters and more hidden layers, to evaluate more complex LSTM topology architectures, that may provide better results.
- The above-mentioned models (CNN, LSTM and models by using all dataset’s features) will be further investigated in a future work study.

Chapter 5. Conclusion - Further Discussion

A single-hidden layer Deep Learning Neural Network model reached a higher weighted F-score (85,2%) than the weighted F-scores of the classical Machine Learning models, as shown in Table 19. Also, the devmax.DF feature selection metric reached higher scores in most Classification models. The DL classifier that achieved 85,2 % weighted F-score consists of a one-hidden layer DL NN, with a volume of hidden units equal to the input layer's size multiplied by 45 and by using a dropout layer of 90% chance.

Table 19: Results of DL & Classical ML models for text classification

Classifier	metric	Vector Size	wPrecision	wRecall	wF-score
NB	x^2	200	76,6%	69,9%	72,8%
k-NN	Devmax.DF	100	57,6%	52,9%	52,5%
SVM	x^2	200	85,2%	79,5%	81,7%
RF	Devmax.DF	100	89,5%	70,8%	78,3%
LR	Devmax.DF	300	87,4%	78,6%	82,6%
DL (simple)[20]	Devmax.DF	300	84,1%	78,7%	81,1%
DL[21]	**Devmax.DF**	**300**	**86,8%**	**84,7%**	**85,2%**

Though a single-hidden layer of NN DL performed better on this specific dataset, the findings cannot be further generalized. Further research is needed with numerous datasets, and different types of datasets (binary, multiclass, multilabel, multioutput). The same applies for the findings related to devmax.DF feature selection metric. Devmax.DF outperformed tf.idf and x^2, but further research is needed with more datasets to confirm devmax.DF's potential.

Also, different tests were performed by using all dataset's features with a simple DL NN, a simple CNN model and a simple LSTM model (see pages 48-50). The results indicate that there might be a potential for higher F-scores. Nevertheless, further tests are needed with more complex topologies and different hyperparameters. These experiments will be performed in a future work study.

Lastly, it is possible that the models have overfitted the dataset. Other techniques are needed, such as "simple hold-out validation" method, to verify the models' generalization capabilities.

[20] Results from Table 11. This F-score was achieved with a simple DL NN.

[21] The highest weighted F-score is shown in Table 14.

Bibliography

Abadi, M., Agarwal, A., Barham, P., & others. (2016). TensorFlow: A system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design.*

Altman, N. S. (1992). An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician, 46.3*, 175-185.

Berrar, D. (2018). Retrieved from Research Gate: https://www.researchgate.net/publication/324701535_Cross-Validation

Bird, Steven, Loper, E., & Klein, E. (2009). *Natural Language Processing with Python.* O'Reilly Media Inc.

Bisong, E. (2019). *Google Colaboratory. In: Building Machine Learning and Deep Learning Models on Google Cloud Platform.* CA: Apress, Berkeley. doi:https://doi.org/10.1007/978-1-4842-4470-8_7

Breiman, L., Friedman, H. J., Olshen, A. R., & Stone, J. C. (1984). *Classification And Regression Trees.* New York: Routledge.

Buitinck, L., & others. (2013). API design for machine learning software: experiences from the scikit-learn project.

Chollet, F. (2017). *Deep Learning with Python MEAP.* Manning Publications.

Chollet, F., & others, &. (2015). *GitHub*. Retrieved from https://github.com/fchollet/keras

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, 861-874.

Geron, A. (2019). *Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow.* O'Reilly.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* London: MIT Press.

Gunasekara, S. S., & Haddela, P. S. (2018). Context aware stopwords for Sinhala Text classification. *National Information Technology Conference (NITC).* Colombo, Sri Lanka.

Harris, C., Millman, K., van der Walt, S., & others. (2020). Array programming with NumPy. *Nature*, 357–362.

Hastie, T., Friedman, J., & Tibshirani, R. (2001). *The Elements of Statistical Learning.* Springer.

Heaton, J. (2008). *Introduction to Neural Networks with Java.* Heaton Research, Inc.

Ho, T. K. (1995). Random decision forests. In Proceedings of 3rd international conference on document analysis and recognition., *1*, pp. 278-282.

Jivani, A. G. (2011). A Comparative Study of Stemming Algorithms. *Anjali Ganesh Jivani et al, Int. J. Comp. Tech. Appl., 2*, 1930-1938.

Kaur, J., & Buttar, P. (2018, April). A Systematic Review on Stopword Removal Algorithms. *International Journal on Future Revolution in Computer Science & Communication Engineering, 4*(4), 207-210.

Kubat, M. (2017). *An Introduction to Machine Learning.* Springer.

Ladani, S. J., & Desai, N. P. (2020). Stopword Identification and Removal Techniques on TC and IR applications: A Survey. *6th International Conference on Advanced Computing & Communication Systems (ICACCS)*, (pp. 466-472).

Lane, H., Howard, C., & Hapke, H. (2019). *Natural Language Processing in Action.* Manning.

Langey, P. (1988). Machine Learning as an Experimental Science. *Machine Learning*, 3-8.

Muller, K.-R., Mika, S., Ratsch, G., Tsuda, K., & Scholkopf, B. (2001). An Introduction to Kernel-based Learning Algorithms. *IEEE Transactions on Neural Networks*, 181-201.

North, M. (2012). *Data mining for the masses.* Global Text.

Pedregosa, F., & others. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 2825-2830.

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach.* Malaysia: Pearson Education Limited.

Samuel, A. L. (1959, July). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development, 3*(3), 210-229.

Sarker, I. (2021). Machine Learning: Algorithms, Real-World Applications and Research Directions. *SN COMPUT. SCI.* doi:https://doi.org/10.1007/s42979-021-00592-x

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction.* MIT press.

Triantafyllou, I., Drivas, I. C., & Giannakopoulos, G. (2020). How to Utilize my App Reviews? A Novel Topics Extraction Machine Learning Schema for Strategic Buisness Purposes. *entropy*.

Vajjala, S., Majumder, B., Gupta, A., & Surana, H. (2020). *Practical Natural Language Processing.* O'Reilly.

Webb, G. (2011). *Encyclopedia of Machine Learning.* Boston, MA: Springer.

Webster, J., & Kit, C. (1992). Tokenization as the initial phase in NLP. *Proceedings of the 14th conference on Computational linguistics*, *4*, pp. 1106-1110. Nantes.

Appendix

Appendix – A

In the table below, you can find the results of testing different DL architecture-topologies (schema), with different number of hidden layers, and with the use of different feature selection metrics, namely chi square and devmax.DF.

Schema	Metric	Hidden Layers	wPrecision	wRecall	wFscore
>	chi square	2	85,9%	80,9%	82,2%
>	chi square	3	86,4%	80,4%	81,9%
>	chi square	4	86,0%	79,6%	81,1%
>	chi square	5	86,2%	78,2%	80,2%
>	chi square	6	85,2%	76,4%	78,4%
>	chi square	7	84,4%	74,9%	76,8%
>	chi square	8	81,8%	73,2%	74,7%
>	chi square	9	78,2%	70,7%	72,3%
>	chi square	10	73,4%	68,2%	69,9%
>	chi square	11	71,5%	67,9%	69,2%
>	chi square	12	71,4%	65,4%	67,3%
>	chi square	13	69,6%	62,2%	64,3%
>	chi square	14	61,3%	56,3%	57,6%
>	chi square	15	33,4%	40,6%	30,4%
<	chi square	2	85,9%	80,9%	82,2%
<	chi square	3	85,9%	80,4%	81,8%
<	chi square	4	86,2%	80,1%	81,6%
<	chi square	5	86,3%	78,7%	80,6%
<	chi square	6	85,8%	77,5%	79,4%
<	chi square	7	84,9%	77,9%	79,7%
<	chi square	8	84,5%	75,9%	77,8%
<	chi square	9	84,2%	75,5%	77,4%
<	chi square	10	82,2%	73,5%	75,4%
<	chi square	11	80,2%	72,0%	73,7%
<	chi square	12	76,4%	70,5%	71,9%
<	chi square	13	74,4%	69,0%	70,2%
<	chi square	14	72,8%	68,4%	69,8%
<	chi square	15	69,4%	65,0%	65,6%
><	chi square	4	86,0%	79,3%	80,9%
><	chi square	6	85,8%	77,2%	79,1%
><	chi square	8	83,6%	73,9%	75,6%
><	chi square	10	76,0%	69,7%	71,2%
><	chi square	12	73,0%	67,8%	69,3%
><	chi square	14	68,9%	64,3%	65,6%
><	chi square	16	32,5%	38,0%	27,4%
><	chi square	18	23,2%	32,5%	15,5%
><	chi square	20	14,3%	31,9%	13,8%
><	chi square	22	10,4%	32,8%	13,6%
><	chi square	24	11,0%	32,7%	13,6%
><	chi square	26	10,4%	32,9%	13,4%
><	chi square	28	14,7%	33,9%	15,0%
><	chi square	30	11,5%	34,0%	15,0%
<>	chi square	4	85,9%	79,6%	81,2%
<>	chi square	6	85,3%	78,6%	80,1%
<>	chi square	8	85,2%	75,2%	77,2%
<>	chi square	10	80,7%	72,1%	74,0%
<>	chi square	12	72,8%	68,1%	69,8%
<>	chi square	14	71,9%	66,6%	67,9%
<>	chi square	16	64,1%	59,2%	59,9%
<>	chi square	18	33,3%	38,9%	29,9%
<>	chi square	20	20,6%	25,8%	16,0%
<>	chi square	22	12,3%	31,4%	15,0%
<>	chi square	24	13,9%	32,3%	13,4%
<>	chi square	26	9,1%	32,6%	13,2%
<>	chi square	28	9,3%	32,5%	13,1%
<>	chi square	30	8,8%	32,8%	13,2%
>>	chi square	4	86,4%	79,6%	81,3%
>>	chi square	6	85,8%	77,4%	79,5%
>>	chi square	8	83,6%	74,3%	76,0%
>>	chi square	10	75,7%	69,6%	71,2%
>>	chi square	12	72,8%	67,9%	69,7%
>>	chi square	14	67,7%	63,0%	64,3%
>>	chi square	16	29,0%	38,8%	24,8%
>>	chi square	18	14,6%	32,5%	14,7%
>>	chi square	20	12,2%	32,7%	13,4%
>>	chi square	22	14,2%	32,5%	14,2%
>>	chi square	24	13,3%	33,4%	14,1%
>>	chi square	26	12,6%	32,8%	13,4%
>>	chi square	28	10,6%	34,6%	15,4%
>>	chi square	30	11,2%	33,0%	14,2%
<<	chi square	4	86,2%	80,0%	81,5%
<<	chi square	6	86,0%	77,4%	79,4%
<<	chi square	8	84,3%	76,8%	78,7%
<<	chi square	10	79,8%	71,7%	73,4%
<<	chi square	12	75,4%	70,2%	71,4%
<<	chi square	14	72,3%	67,9%	69,2%
<<	chi square	16	60,5%	57,7%	57,3%
<<	chi square	18	28,8%	25,4%	20,7%
<<	chi square	20	22,6%	27,0%	20,1%
<<	chi square	22	14,9%	30,1%	17,4%
<<	chi square	24	10,8%	30,6%	14,3%
<<	chi square	26	10,1%	32,3%	13,7%
<<	chi square	28	10,2%	30,2%	13,1%
<<	chi square	30	9,0%	30,9%	13,1%
|||	chi square	2	86,0%	80,7%	82,1%
|||	chi square	3	86,1%	80,7%	82,1%
|||	chi square	4	86,3%	80,4%	81,8%
|||	chi square	5	86,4%	78,0%	80,3%
|||	chi square	6	85,7%	79,0%	80,5%
|||	chi square	7	84,8%	76,6%	78,4%
|||	chi square	8	84,9%	76,2%	78,2%
|||	chi square	9	83,5%	75,6%	77,3%
|||	chi square	10	78,5%	71,2%	72,9%
|||	chi square	11	75,8%	68,2%	70,3%
|||	chi square	12	72,2%	67,7%	69,2%
|||	chi square	13	71,6%	67,2%	68,6%
|||	chi square	14	70,9%	67,9%	68,8%
|||	chi square	15	59,8%	57,7%	56,8%
|	chi square	1	86,0%	81,9%	82,9%
>	devmax.DF	2	85,7%	84,0%	84,2%
>	devmax.DF	3	85,8%	83,6%	83,9%
>	devmax.DF	4	85,6%	83,2%	83,5%
>	devmax.DF	5	85,2%	81,6%	82,3%
>	devmax.DF	6	85,3%	80,6%	81,7%
>	devmax.DF	7	85,2%	79,0%	80,5%
>	devmax.DF	8	85,0%	79,4%	80,8%
>	devmax.DF	9	81,4%	75,3%	76,7%
>	devmax.DF	10	77,9%	72,4%	73,6%
>	devmax.DF	11	75,5%	70,4%	71,7%
>	devmax.DF	12	72,3%	67,0%	68,9%
>	devmax.DF	13	69,9%	66,7%	67,7%
>	devmax.DF	14	66,3%	61,6%	62,9%
>	devmax.DF	15	59,5%	55,8%	56,7%
<	devmax.DF	2	85,6%	84,0%	84,2%
<	devmax.DF	3	85,7%	83,8%	84,0%
<	devmax.DF	4	85,6%	83,4%	83,7%
<	devmax.DF	5	85,9%	81,7%	82,7%
<	devmax.DF	6	85,2%	81,8%	82,5%
<	devmax.DF	7	84,8%	80,5%	81,1%
<	devmax.DF	8	84,7%	80,8%	81,7%
<	devmax.DF	9	84,6%	76,4%	78,4%
<	devmax.DF	10	83,9%	76,5%	78,6%
<	devmax.DF	11	83,8%	77,3%	79,2%
<	devmax.DF	12	80,6%	74,1%	76,2%
<	devmax.DF	13	75,2%	70,8%	72,0%
<	devmax.DF	14	75,0%	69,3%	70,8%
<	devmax.DF	15	74,0%	69,4%	70,6%
><	devmax.DF	4	86,1%	82,6%	83,3%
><	devmax.DF	6	85,2%	81,4%	82,2%
><	devmax.DF	8	85,2%	80,3%	81,6%
><	devmax.DF	10	78,5%	73,5%	74,8%
><	devmax.DF	12	73,1%	69,5%	70,3%
><	devmax.DF	14	71,2%	68,1%	68,8%
><	devmax.DF	16	41,4%	45,7%	39,2%
><	devmax.DF	18	26,9%	40,1%	25,6%
><	devmax.DF	20	21,5%	28,1%	16,9%
><	devmax.DF	22	10,8%	29,4%	13,7%
><	devmax.DF	24	9,9%	32,6%	13,4%

><	devmax.DF	26	10,1%	32,7%	13,3%
><	devmax.DF	28	11,2%	34,0%	14,9%
><	devmax.DF	30	10,6%	32,8%	14,2%
<>	devmax.DF	4	85,9%	82,9%	83,6%
<>	devmax.DF	6	85,1%	81,5%	82,2%
<>	devmax.DF	8	84,5%	80,0%	80,9%
<>	devmax.DF	10	84,0%	77,6%	79,3%
<>	devmax.DF	12	77,8%	72,4%	74,2%
<>	devmax.DF	14	74,2%	69,3%	70,6%
<>	devmax.DF	16	69,7%	67,7%	68,3%
<>	devmax.DF	18	45,7%	49,9%	44,8%
<>	devmax.DF	20	27,8%	35,5%	26,4%
<>	devmax.DF	22	21,8%	30,1%	19,3%
<>	devmax.DF	24	13,6%	30,1%	15,0%
<>	devmax.DF	26	9,3%	31,8%	14,0%
<>	devmax.DF	28	8,5%	32,7%	13,1%
<>	devmax.DF	30	8,4%	32,9%	13,4%
>>	devmax.DF	4	85,5%	82,8%	83,2%
>>	devmax.DF	6	85,3%	81,0%	81,8%
>>	devmax.DF	8	84,6%	79,9%	80,9%
>>	devmax.DF	10	78,7%	72,2%	73,8%
>>	devmax.DF	12	73,6%	68,5%	70,0%
>>	devmax.DF	14	70,3%	65,6%	67,4%
>>	devmax.DF	16	46,0%	49,9%	44,4%
>>	devmax.DF	18	28,7%	34,7%	20,1%
>>	devmax.DF	20	16,2%	32,1%	15,1%
>>	devmax.DF	22	13,6%	29,2%	12,6%
>>	devmax.DF	24	13,5%	32,8%	13,7%
>>	devmax.DF	26	11,4%	32,6%	13,3%
>>	devmax.DF	28	13,4%	32,8%	13,6%
>>	devmax.DF	30	9,8%	32,9%	14,1%
<<	devmax.DF	4	85,7%	83,4%	83,8%
<<	devmax.DF	6	85,1%	80,7%	81,7%
<<	devmax.DF	8	84,7%	79,6%	80,9%
<<	devmax.DF	10	84,7%	78,0%	80,1%
<<	devmax.DF	12	76,1%	70,5%	72,1%
<<	devmax.DF	14	72,3%	69,3%	70,2%
<<	devmax.DF	16	68,1%	65,6%	65,2%
<<	devmax.DF	18	44,9%	43,2%	39,5%
<<	devmax.DF	20	31,2%	27,1%	25,2%
<<	devmax.DF	22	22,0%	27,0%	18,0%
<<	devmax.DF	24	17,7%	26,1%	17,7%
<<	devmax.DF	26	15,2%	29,4%	17,4%
<<	devmax.DF	28	10,0%	31,0%	14,5%
<<	devmax.DF	30	12,0%	33,1%	15,8%
|||	devmax.DF	2	85,6%	83,9%	84,1%
|||	devmax.DF	3	86,1%	83,9%	84,2%
|||	devmax.DF	4	85,9%	83,4%	83,7%
|||	devmax.DF	5	85,8%	81,8%	82,8%
|||	devmax.DF	6	85,5%	81,1%	82,2%
|||	devmax.DF	7	84,8%	80,3%	81,3%
|||	devmax.DF	8	84,8%	79,3%	80,6%
|||	devmax.DF	9	83,8%	78,5%	79,9%
|||	devmax.DF	10	83,4%	75,6%	77,4%
|||	devmax.DF	11	79,3%	72,9%	74,2%
|||	devmax.DF	12	75,7%	70,3%	71,3%
|||	devmax.DF	13	73,8%	70,1%	71,1%
|||	devmax.DF	14	71,3%	67,9%	68,9%
|||	devmax.DF	15	70,7%	67,7%	68,7%
|	devmax.DF	1	86,2%	84,6%	84,9%

Appendix – B

In the table below you can find the results of testing a simple one-hidden layer Deep Learning Neural Network, with different sizes of hidden layer's units. The multiplier increases the initial size, by 1, 2, 3, … 200 times of the Lexicon's length (=300) that was used in the Bag of Words. So, for example the hidden layer with a multiplier 5, is referring to a hidden layer with size = 300*5 = 1500

multiplier	wPrecision	wRecall	wFscore
1	86,2%	84,6%	84,9%
2	86,1%	84,6%	84,8%
3	85,9%	84,7%	84,8%
4	85,8%	85,1%	85,0%
5	85,5%	84,9%	84,8%
6	85,5%	85,0%	84,8%
7	85,4%	85,1%	84,8%
8	85,5%	85,0%	84,8%
9	85,8%	85,1%	85,0%
10	85,8%	84,7%	84,8%
11	85,6%	84,9%	84,8%
12	85,5%	85,1%	84,9%
13	85,4%	84,9%	84,7%
14	85,6%	85,1%	84,9%
15	85,2%	85,1%	84,7%
16	84,9%	85,3%	84,7%
17	85,0%	85,2%	84,6%
18	85,3%	85,2%	84,8%
19	85,0%	85,2%	84,7%
20	85,1%	85,2%	84,7%
21	85,3%	85,3%	84,9%
22	85,2%	85,1%	84,7%
23	85,4%	85,2%	84,9%
24	85,3%	85,0%	84,7%
25	85,0%	85,1%	84,6%
26	85,3%	85,0%	84,8%
27	85,2%	85,1%	84,7%
28	85,1%	85,1%	84,7%
29	85,4%	85,1%	84,8%
30	85,8%	84,5%	84,7%
31	85,7%	84,5%	84,6%
32	85,5%	84,5%	84,5%
33	85,6%	84,3%	84,5%
34	85,8%	84,7%	84,7%
35	85,8%	84,4%	84,6%
36	85,5%	84,4%	84,5%
37	85,6%	84,5%	84,6%
38	85,5%	84,7%	84,6%
39	85,6%	84,7%	84,7%
40	85,5%	84,6%	84,6%
41	85,5%	84,8%	84,6%
42	85,3%	84,8%	84,5%
43	85,8%	84,5%	84,7%
44	85,7%	84,7%	84,7%
45	85,4%	84,5%	84,4%
46	85,2%	84,5%	84,4%
47	85,7%	84,3%	84,5%
48	85,4%	84,6%	84,6%
49	85,4%	84,5%	84,5%
50	85,5%	84,4%	84,5%
51	85,5%	84,6%	84,6%
52	84,9%	84,9%	84,4%
53	85,6%	84,7%	84,7%
54	85,4%	84,7%	84,5%
55	85,7%	84,6%	84,7%
56	85,4%	84,7%	84,6%
57	85,3%	84,5%	84,5%
58	85,6%	84,5%	84,6%
59	85,6%	84,3%	84,5%
60	85,4%	84,8%	84,7%
61	85,4%	84,7%	84,5%
62	85,6%	84,4%	84,5%
63	85,1%	84,7%	84,4%
64	85,4%	84,5%	84,5%
65	85,4%	84,7%	84,6%
66	85,5%	84,7%	84,7%
67	85,2%	84,8%	84,6%
68	85,3%	84,8%	84,6%
69	85,3%	84,5%	84,5%
70	84,9%	84,8%	84,4%

71	85,5%	84,3%	84,4%
72	85,4%	84,6%	84,5%
73	85,2%	84,8%	84,5%
74	85,6%	84,3%	84,5%
75	85,4%	84,7%	84,6%
76	85,1%	84,8%	84,5%
77	85,1%	84,6%	84,4%
78	85,2%	84,3%	84,3%
79	85,1%	84,9%	84,6%
80	85,3%	84,4%	84,4%
81	85,0%	84,8%	84,5%
82	85,0%	84,9%	84,5%
83	84,9%	84,6%	84,3%
84	85,1%	84,4%	84,3%
85	85,2%	84,3%	84,3%
86	85,0%	84,5%	84,3%
87	84,9%	84,5%	84,3%
88	85,2%	84,9%	84,6%
89	85,1%	84,5%	84,3%
90	85,1%	84,7%	84,4%
91	85,4%	84,8%	84,6%
92	85,4%	84,4%	84,5%
93	84,8%	84,8%	84,4%
94	85,2%	85,0%	84,6%
95	85,3%	84,5%	84,5%
96	85,2%	84,7%	84,5%
97	84,9%	84,9%	84,5%
98	85,0%	84,7%	84,4%
99	85,1%	84,4%	84,3%
100	85,5%	84,2%	84,4%
101	84,4%	85,5%	84,6%
102	84,6%	85,0%	84,4%
103	84,8%	85,2%	84,6%
104	84,8%	85,3%	84,7%
105	84,8%	85,2%	84,6%
106	85,2%	85,2%	84,8%
107	85,0%	85,1%	84,6%
108	84,7%	85,1%	84,5%
109	85,0%	84,9%	84,5%
110	84,5%	85,2%	84,4%
111	84,9%	85,0%	84,5%
112	84,8%	85,0%	84,5%
113	85,1%	85,0%	84,6%
114	84,9%	85,1%	84,6%
115	84,7%	85,1%	84,5%
116	84,4%	85,0%	84,3%
117	84,8%	85,3%	84,6%
118	85,1%	85,2%	84,7%
119	85,0%	84,9%	84,6%
120	85,0%	84,8%	84,5%
121	84,7%	85,2%	84,6%
122	84,7%	85,2%	84,5%
123	85,1%	85,1%	84,7%
124	84,8%	85,0%	84,5%
125	84,9%	85,2%	84,6%
126	84,3%	85,3%	84,4%
127	84,9%	85,1%	84,5%
128	84,7%	84,7%	84,3%
129	84,4%	85,2%	84,4%
130	84,6%	84,9%	84,3%
131	84,3%	85,2%	84,4%
132	85,1%	84,9%	84,6%
133	84,9%	85,1%	84,6%
134	84,5%	85,0%	84,3%
135	84,6%	85,1%	84,5%
136	84,7%	84,9%	84,4%
137	84,6%	85,0%	84,4%
138	84,9%	85,0%	84,6%
139	84,7%	85,3%	84,6%
140	85,0%	84,9%	84,6%
141	84,2%	85,2%	84,3%
142	84,7%	85,1%	84,5%
143	84,3%	85,2%	84,3%
144	84,9%	84,8%	84,4%
145	84,9%	84,7%	84,4%
146	84,3%	85,0%	84,3%
147	84,5%	85,0%	84,4%
148	84,3%	85,0%	84,2%
149	84,8%	85,3%	84,7%
150	84,7%	85,1%	84,5%
151	84,4%	84,9%	84,3%
152	84,2%	85,1%	84,2%
153	84,2%	85,2%	84,2%
154	84,7%	85,0%	84,5%
155	84,3%	85,0%	84,3%
156	84,6%	85,1%	84,5%
157	85,0%	85,1%	84,7%
158	84,0%	85,3%	84,2%
159	84,7%	84,9%	84,3%
160	84,4%	85,1%	84,4%
161	84,7%	85,2%	84,6%
162	84,8%	84,9%	84,4%
163	84,6%	84,8%	84,2%
164	84,6%	85,0%	84,4%
165	84,8%	85,1%	84,6%
166	84,4%	85,2%	84,4%
167	84,5%	85,0%	84,3%
168	84,0%	85,3%	84,3%
169	85,1%	85,0%	84,6%
170	84,3%	85,2%	84,3%
171	84,1%	85,1%	84,2%
172	84,4%	85,0%	84,3%
173	84,6%	84,8%	84,3%
174	84,8%	84,6%	84,3%
175	84,9%	84,9%	84,5%
176	84,3%	85,1%	84,3%
177	84,2%	85,1%	84,3%
178	84,8%	84,9%	84,4%
179	84,3%	85,4%	84,4%
180	84,4%	85,1%	84,4%
181	84,2%	84,9%	84,1%
182	84,5%	85,2%	84,4%
183	84,2%	85,0%	84,2%
184	84,2%	85,1%	84,3%
185	84,4%	85,1%	84,3%
186	84,5%	84,9%	84,3%
187	84,3%	85,3%	84,4%
188	84,3%	85,3%	84,4%
189	84,6%	84,5%	84,1%
190	84,3%	85,4%	84,5%
191	84,4%	84,9%	84,3%
192	84,4%	85,0%	84,4%
193	84,2%	85,4%	84,4%
194	84,2%	85,4%	84,4%
195	84,2%	84,9%	84,1%
196	84,6%	84,6%	84,2%
197	84,6%	84,8%	84,3%
198	84,0%	85,2%	84,2%
199	84,7%	84,7%	84,3%
200	84,4%	84,8%	84,2%

Appendix – C

The function below (created with Python language) checks if two words are similar, given some criteria. More specifically, this function:

1. receives two parameters, w1 and w2
2. calculates their average length and assigns the result to array L
3. calculates the 66% (2/3) of the average length, and assigns the result to array match
4. checks if w1 and w2 have both a length over four
5. checks if the left part of w1 and w2 (from the first character until the number that is stored in the match variable) are exactly the same
6. If steps 4 and 5 are True, then the function returns True, otherwise it returns False.
7. The method math.ceil() rounds a float to the upper integer and returns an integer.

```python
def CalculateWordSimilarity(w1, w2):

    L = (len(w1) + len(w2)) / 2

    match = math.ceil(L * 0.66)

    return w1[:match] == w2[:match] and len(w1) > 4 and len(w2) > 4
```

Appendix – D

The function below (created with Python language) transforms text into tokens. More specifically it:

1. receives the file path of a csv file
2. reads each line of the csv files and append the data to an array All_Data
3. Parses the All_Data array and:
    a. performs tokenization by using regular expressions
    b. lowers all words' letters
    c. stems the words
    d. ignores stop word of the English language
4. stores and returns the tokenized data

```python
def PrePrepocessing(filepath):

# the following library is imported so to remove punctuation with regex ruels
    from nltk.tokenize import RegexpTokenizer
    # the following library is imported so to ignore stopwords
    from nltk.corpus import stopwords
    # the following library is imported so to stem words
    from nltk import PorterStemmer
    ps = PorterStemmer()

    # opens csv and extracts all data into one array: All_Data
    All_Data = []
    with open(filepath, newline='', encoding='utf8') as csvfile:
        for row in csv.reader(csvfile):
            All_Data.append(row)

    #sets the rule to ignore stopwords
    tokenizer = RegexpTokenizer(r'\w+(?:-\w+)*')

    # defines the following array and adds the headers of All_Data
    Normalized_Data = []

    # performs tokenization, stems, uppers words and ignores stopwords
    for row in All_Data[1:]:
        temp_row = []
        for token in tokenizer.tokenize(row[0].replace("'", "")):

            token = token.lower()
            token = ps.stem(token)

            if token not in stopwords.words('english'):
                temp_row.append(token)
        Normalized_Data.append(temp_row)

    # saves the Normalized_Data array into the project's directory
    with open('Normalized_Data_file', 'wb') as fp:
        pickle.dump(Normalized_Data, fp)
```

Appendix – E

The code below (created with Python language) creates a Python dictionary with all words that appear in the given data. More specifically:

1. It iterates through the saved data
2. It ignores words with a length of 1
3. It adds in the dictionary a word as the dictionary's key. In parallel, it counts the term frequency and the frequency in each category for the specific word.
4. Right after, for each word in Lexicon in calculates the sum of the frequency it appears in each category, and the document frequency

```
Lexicon = {}
# reads each row of Normalized_Data and adds keys-items in the Lexicon
for item in Normalized_Data[1:]:
    for word in item[0]:
        # ignores words with len=1
        if len(word) == 1:
            continue
        # if word is in Lexicon then it just adds the 1
        if word in Lexicon.keys():
            Lexicon[word]['tf'] += 1
            for i in range(12):
                Lexicon[word][i] += int(item[1][i])
        # if word is not in Lexicon, it creates the key and adds the metadata
        else:
            Lexicon[word] = {'tf': 1, 'df': 0, 'Sum_cat': 0, 'len': len(word)}
            for i in range(12):
                Lexicon[word][i] = int(item[1][i])

for word in Lexicon.keys():
    Lexicon[word]['Sum_cat'] = sum(list(Lexicon[word].values())[4:])
    # calculates the document frequency
    for item in Normalized_Data[1:]:
        if word in item[0]:
            Lexicon[word]['df'] += 1
```

Appendix – F

The code below (created with Python language) iterates through different Lexicon sizes and through some classical ML models and one simple DL model. This procedure saved time. Instead of running multiple times the script, with this iteration it run all models in a one-off procedure.

```
def RunModels(Normalized_Data, batch_index, AllLexicons, Results, method):
    Lexicon = AllLexicons
    for LexiconSize in [100, 200, 300, 400]:


        Temp_Lexicon = [i[0] for i in Lexicon[:LexiconSize]]
        X_train = []

        for item in Normalized_Data[1:]:
            temp = [0] * LexiconSize
            for word in item[0]:
                if word in Temp_Lexicon:
                    temp[Temp_Lexicon.index(word)] = 1
                    continue
             X_train.append(temp)

        for i in [0, 1, 2, 3, 4, 5, 6]:
            rc = []
            rc.append(batch_index)
            rc.append(str(datetime.datetime.now()))
            rc.append(featureExtractionMethod(method))
            rc.append(LexiconSize)
            if i == 0:
                # runs the Naive Bayes model
                print("NB... loading")
                rc = sklearnmodels("NB", sklearn.naive_bayes.MultinomialNB(), X_train,
Normalized_Data, rc)
            elif i == 1:
                # runs the kNN model
                print("kNN... loading")
                rc = sklearnmodels("kNN", KNeighborsClassifier(n_neighbors=1),
X_train, Normalized_Data, rc)
            elif i == 2:
                # runs the SVM model
                print("SVM... loading")
                rc = sklearnmodels("SVM",
                                   make_pipeline(StandardScaler(), SVC(C=1,
kernel='linear', degree=3, gamma='auto')),
                                   X_train, Normalized_Data, rc)
            elif i == 3:
                # runs the Random Forest model
                print("RF... loading")
                rc = sklearnmodels("RF", RandomForestClassifier(max_depth=15,
random_state=1), X_train, Normalized_Data,
                                   rc)
            elif i == 4:
                # runs the Logistic Regression Classifier
                rc = sklearnmodels("LR", LogisticRegression(), X_train,
Normalized_Data, rc)
                print("LR... loading")
            elif i == 5:
                # runs the Multi-Layer P Classifier (= Deep Learning)
                print("DL... loading")
                rc = sklearnmodels("DL",
                                   MLPClassifier(solver='adam', activation='relu',
batch_size=32, n_iter_no_change=20,
                                                 hidden_layer_sizes=(100,),
random_state=0, early_stopping=True),
                                   X_train, Normalized_Data, rc)

            Results.append(rc)
```

Appendix – G

The code below (created with Python language) activates the sklearnmodels() function that is called in the code of Appendix – F. For each category it runs the model. The y_train data is created ad hoc, because not all models are capable of running multilabel classification. Thus, the models run in a binary form. In the end it returns the results of the iteration.

```
def sklearnmodels(ML_model, cv_model, X_train, Normalized_Data, rc):
    start_time = time.time()
    y_pred = [None] * 12
    y_train = [None] * 12
    # loops through the 12 categories of the datasets
    for k in range(12):
        # extracts the k category of the targets, to train the model
        y_train[k] = [int(i[1][k]) for i in Normalized_Data[1:]]
        y_pred[k] = list(cross_val_predict(cv_model, np.array(X_train), y_train[k],
cv=10))

    new_y_train = []
    new_y_pred = []

    for i in range(len(y_train[0])):
        temp_ytrain = []
        temp_ypred = []
        for k in range(12):
            temp_ytrain.append(y_train[k][i])
            temp_ypred.append(y_pred[k][i])
        new_y_train.append(temp_ytrain)
        new_y_pred.append(temp_ypred)


    rc.append(ML_model)

    acc = []
    for i in range(12):
        acc.append(round(accuracy_score([k[i] for k in new_y_train], [k[i] for k in
new_y_pred]), 4))

    rc.append(round(mean(acc), 4))
    results1 = precision_recall_fscore_support(new_y_train, new_y_pred,
average='weighted')
    rc.append(round(results1[0], 4))
    rc.append(round(results1[1], 4))
    rc.append(round(results1[2], 4))
    rc.append(round(rc[-3] * rc[-1], 4))
    rc.append(round(int(time.time() - start_time)))
    print(rc)
    print(f1_score(new_y_train, new_y_pred, average='weighted'))
    return rc
```

Appendix – H

The code below (created with Python language), which is part of a bigger amount of code, runs and saves the data of a "pyramid" DL NN model. The k value is increasing in each iteration, adding more layers in each iteration. In Parallel, the more layers are added the fewer are the hidden layer's units in each iteration (-5%). Also, it runs the model 10 times, one for each fold.

```
for k in range(1, 15):
    Fold10Averages = []
    list10fold = [[0, 0.1], [0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5], [0.5,
0.6], [0.6, 0.7], [0.7, 0.8], [0.8, 0.9], [0.9, 1]]
    for fold in list10fold:
        X_train, y_train, X_test, y_test = ReturnTrainTest(fold, All_Data_Normalized)
        metric = LexiconSize
        model = models.Sequential()
        model.add(layers.Dense(newmetric, activation='selu', input_shape=(metric,)))
        model.add(layers.Dropout(0.3))
        for i in range(k):
            newmetric = int(newmetric * 0.95)
            model.add(layers.Dense(newmetric, activation='selu'))
            model.add(layers.Dropout(0.3))
        Fold10Averages.append(RunModel(model, X_train, y_train, X_test, y_test))
    CalculateAverage(Fold10Averages, 2,  "pyramid >" + str(k), strmetric)
```

The code below (created with Python language) runs the function ReturnTrainTest() that is called in the above code. It returns the corresponding fold that need to be trained-tested in the DL mode.

```
def ReturnTrainTest(i, All_Data_Normalized):
    lnght = len(All_Data_Normalized[0])
    rnd = [floor(i[0] * lnght), floor(i[1] * lnght)]
    X_train = All_Data_Normalized[0][0:rnd[0]] + All_Data_Normalized[0][rnd[1]:]
    y_train = All_Data_Normalized[1][0:rnd[0]] + All_Data_Normalized[1][rnd[1]:]
    X_test = All_Data_Normalized[0][rnd[0]:rnd[1]]
    y_test = All_Data_Normalized[1][rnd[0]:rnd[1]]
    assert len(X_train) + len(X_test) == lnght
    return X_train, y_train, X_test, y_test
```

The code below (created with Python language), which is part of a bigger amount of code, runs the function RunModel() that is called in the first code presented in Appendix – H. It adds the last layer (output layer), compiles the model, fits it, and created predictions.

```
def RunModel(model, X_train, y_train, X_test, y_test):
    callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=7,
mode='auto')
    classweights = {0: 178, 1: 273, 2: 513, 3: 590, 4: 141, 5: 290, 6: 733, 7: 254, 8:
1934, 9: 261, 10: 166, 11: 574}
    model.add(layers.Dense(12, activation='sigmoid', use_bias=True))  # 12 because we
have 12 categories
    model.compile(optimizer=Adam(learning_rate=0.0001), loss='binary_crossentropy',
metrics=['accuracy'])
   #print(model.summary())
    history = model.fit(x=np.array(X_train), y=np.array(y_train),
validation_split=0.1, batch_size=32,
              epochs=300, shuffle=True, callbacks=[callback],
              verbose=0, class_weight=classweights, use_multiprocessing=True)
    epoxes= len(history.history['val_loss'])
    print(epoxes, "epochs")
    predictions = model.predict(x=np.array(X_test), batch_size=32, verbose=0,
use_multiprocessing=True)
```

Appendix – I

The code below (created with Python language), which is part of a bigger amount of code, runs a CNN model in 10-folds.

```
for fold in list10fold:

    X_train, y_train, X_test, y_test = ReturnTrainTest(fold, temp)

    model = Sequential()
    model.add(Embedding(300, megethos, input_length=megethos))
    model.add(Conv1D(1500, 4, padding='same', activation='relu'))
    model.add(GlobalMaxPool1D())
    callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=7,
mode='auto')
    model.add(Dense(12, activation='sigmoid', use_bias=True))
    model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy',
metrics=['accuracy'])

    history = model.fit(x=np.array(X_train), y=np.array(y_train),
validation_split=0.1, batch_size=300,
                        epochs=300, shuffle=True, callbacks=[callback],
                        verbose=0, class_weight=classweights,
use_multiprocessing=True)
    predictions = model.predict(x=np.array(X_test), batch_size=32, verbose=0,
use_multiprocessing=True)
```