UNIVERSITY OF THE PELOPONNESE &

NCSR “DEMOCRITOS”

MSC PROGRAMME IN DATA SCIENCE

Document Data Analysis via Machine/Deep Learning techniques

by

Angelos Spyratos
2022201704021

A thesis submitted in partial fulfillment
of the requirements for the MSc
in Data Science

Supervisor: Georgios Petasis

Athens, January 2021

Document Data Analysis via Machine/Deep Learning techniques
Aggelos Spyratos
MSc. Thesis, MSc. Programme in Data Science
University of the Peloponnese & NCSR Democritos", January 2021

UNIVERSITY OF THE PELOPONNESE &
NCSR “DEMOCRITOS”
MSC PROGRAMME IN DATA SCIENCE

Document Data Analysis via Machine/Deep Learning techniques

by
Aggelos Spyratos
2022201704021

A thesis submitted in partial fulfillment
of the requirements for the MSc
in Data Science

Supervisor: Georgios Petasis

Approved by the examination committee on January, 2021.

Georgios Petasis Anastasia Krithara Konstantinos Vassilakis

……………….. ……………….. ………………..

Athens, January 2021

UNIVERSITY OF THE PELOPONNESE &

NCSR “DEMOCRITOS”

MSC PROGRAMME IN DATA SCIENCE

Declaration of Authorship

(1) I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where stated otherwise by reference or acknowledgment, the work presented is entirely my own.

(2) I confirm that this thesis presented for the degree of Bachelor of Science in Informatics and Telecommunications, has

(i) been composed entirely by myself
(ii) been solely the result of my own work
(iii) not been submitted for any other degree or professional qualification

(3) I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or processional qualification except as specified.

Aggelos Spyratos

Athens, January 2021

ABSTRACT

Job advert aggregators gather millions of adverts every single day, by scraping job boards and various other sources across the globe. Aggregators are getting visited by millions of active job seekers every day, that wish to find their perfect match in order to land a job, according to their skills and field of studies. With such high volume of visitors seeking to find their optimal match, proper categorization of job adverts becomes a must have feature for any aggregator in order to help their users have a smooth experience while searching for their perfect job match. However, due to the huge volume of data and the nature of the job adverts themselves, where each job description can possibly match with multiple categories and similar positions might have huge variations in the language used to describe them, the proper classification of such data comes to be a hard task. In this work, various machine learning, deep learning, data processing and data augmentation methods are used in order to try and classify job adverts in one of the twenty-nine categories of the Adzuna company. Towards this, a real-world private dataset, consisting of about 234.000 job adverts from the United Kingdom, containing titles, descriptions and hand-crafted categories, is provided by the Adzuna company. Our main results show that Deep Learning models outperform all kinds of conventional Machine Learning approaches such as Support Vector Classifiers, Multinomial Naïve Bayes and Decision Trees. In addition, training custom word2vec embeddings helps achieve higher accuracy metrics compared to using pretrained embeddings such as Glove 100. However, the model selection (choosing a Deep Learning model against a conventional Machine Learning model) is of higher impact towards better metrics than using embeddings and sequences of words. The model that achieved the highest weighted average F1-Score (80%) and the highest testing accuracy (80.5%) was the Feedforward Neural Network trained on Bag of Words (TF-IDF) representations of lowercased and stemmed job descriptions. Specifically, this model achieved a weighted average Precision of 80%, a weighted average Recall of 81%.

Keywords: Text Classification, Deep Learning, Machine Learning, Natural Language Processing

LIST OF TABLES

Table 1: The job categories used by Adzuna, serving as labels in the classification task 16
Table 2: Parameters of models used for BoW representations 49
Table 3: Parameters of trained models on sequences of words 50
Table 4: Summary of trained models 51
Table 5: Average metrics of MNB on BoW and lower case descriptions 53
Table 6: Average metrics of MNB on BoW and lower case and stemmed descriptions 55
Table 7: Average metrics of Decision Tree on BoW and lower case and stemmed descriptions 57
Table 8: Average metrics of Decision Tree on BoW and lower case and stemmed descriptions 59
Table 9: Average metrics of SVC on BoW and lower case descriptions 61
Table 10: Average Metrics of SVC on BoW and lower case and stemmed descriptions 63
Table 11: Average Metrics of FNN on Bow and lower case descriptions 65
Table 12: Average Metrics of FNN on BoW and lowercased and stemmed descriptions 67
Table 13: Average Metrics for FNN on Google's word2vec (Glove) embeddings with lower case descriptions 69
Table 14: Average Metrics for FNN custom word2vec embeddings (skipgram) with lower case descriptions 71
Table 15: Average Metrics for FNN custom word2vec embeddings (skipgram) with lower case descriptions 73
Table 16: Average Metrics for CNN on Glove embeddings with lower case descriptions 75
Table 17: Average Metrics for CNN on custom word2vec embeddings with lower case descriptions 77
Table 18: Average Metrics for CNN on custom word2vec embeddings with lower case descriptions 79

Table 19: Average Metrics for RNN on Glove embeddings with lower case descriptions 81
Table 20: Average Metrics for RNN on custom word2vec embeddings with lower case descriptions 83
Table 21: Average Metrics for RNN on custom FastText embeddings with lower case descriptions 85
Table 22: Most common misclassifications among the models 90
Table 22: Python libraries used........ 91

LIST OF FIGURES

Figure 1: An example of transforming text Data to BoW vectors 20
Figure 2: An example of TF-IDF vectors 21
Figure 3: MNB example on text data with alpha value equal to 1 24
Figure 4: A simple example of 2d space and SVM hyperplane with linear kernel 27
Figure 5: Comparison of margin when choosing different margins: Increasing the C parameters, the margin shrinks 28
Figure 6: Activation functions 33
Figure 7: An example architecture of a feedforward multilayer perceptron. 34
Figure 8: Example architecture of a skip-gram training for word2vec embedding..... 37
Figure 9: Example of word2vec representation 37
Figure 10: An example of recurrent architecture 39
Figure 11: A simple recurrent unit in standard RNNs 39
Figure 12: LSTM gate: A sigmoid neural network layer and a pointwise multiplication operation 40
Figure 13: A LSTM cell... 42
Figure 14: An example of convolution with 2x2 filter on 4x4 input and stride length 2 on both axes 44
Figure 15: An example of convolution with 2x2 filter on 3x3 input and stride length 1 on both axes with zero padding 44
Figure 16: Max and Average Pooling over 3x3 convolved feature with 2x2 kernel... 44
Figure 17: CNN architecture example 45
Figure 18: Class distribution of available training and testing Data........................... 46
Figure 19: Classification Report of MNB on BoW and lower case descriptions 54
Figure 20: Classification Report of MNB on BoW and lower case and stemmed descriptions 56
Figure 21: Classification Report of Decision Tree on BoW and lower case descriptions 58
Figure 22: Classification Report of Decision Tree on BoW and lower case and stemmed descriptions 60
Figure 23: Classification Report of SVC on BoW and lower case descriptions 62

Figure 24: Classification Report of SVC on BoW and lower case and stemmed descriptions 64
Figure 25: Loss per training epoch for FNN with lower case descriptions 65
Figure 26: Accuracy per training epoch for FNN with lower case descriptions 65
Figure 27: Classification Report for FNN with lower case descriptions 66
Figure 28: Loss per training epoch for FNN with lower case and stemmed descriptions 67
Figure 29: Accuracy per training epoch for FNN with lower case and stemmed descriptions 67
Figure 30: Classification Report for FNN with lower case and stemmed descriptions 68
Figure 31: Loss per training epoch for FNN on Google's word2vec (Glove) embeddings with lower case descriptions 69
Figure 32: Accuracy per training epoch for FNN on Google's word2vec (Glove) embeddings with lower case descriptions 69
Figure 33: Classification Report for FNN on Google's word2vec (Glove) embeddings with lower case descriptions 70
Figure 34: Loss per training epoch for FNN custom word2vec embeddings (skipgram) with lower case descriptions 71
Figure 35: Accuracy per training epoch for FNN custom word2vec embeddings (skipgram) with lower case descriptions 71
Figure 36: Classification Report for FNN custom word2vec embeddings (skipgram) with lower case descriptions 72
Figure 37: Loss per training epoch for FNN with custom FastText embeddings (skipgram) with lower case descriptions 73
Figure 38: Accuracy per training epoch for FNN with custom FastText embeddings (skipgram) with lower case descriptions 73
Figure 39: Classification Report for FNN with custom FastText embeddings (skipgram) with lower case descriptions 74
Figure 40: Loss per training epoch for CNN with Glove embeddings with lower case descriptions 75

Figure 41: Accuracy per training epoch for CNN with with Glove embeddings with lower case descriptions 75

Figure 42: Classification Report for CNN with Glove embeddings with lower case descriptions 76

Figure 43: Loss per training epoch for CNN custom word2vec embeddings (skipgram) with lower case descriptions 77

Figure 44: Accuracy per training epoch for CNN custom word2vec embeddings (skipgram) with lower case descriptions 77

Figure 45: Classification Report for CNN with custom word2vec embeddings (with skipgram) with lower case descriptions 78

Figure 46: Loss per training epoch for CNN with custom FastText embeddings (skipgram) with lower case descriptions 79

Figure 47: Accuracy per training epoch for CNN with custom FastText word2vec embeddings (skipgram) with lower case descriptions 79

Figure 48: Classification Report for CNN with custom FastText embeddings (with skipgram) with lower case descriptions 80

Figure 49: Loss per training epoch for RNN on Glove embeddings with lower case descriptions 81

Figure 50: Accuracy per training epoch for RNN on Glove embeddings with lower case descriptions 81

Figure 51: Classification Report for RNN on Glove embeddings with lower case descriptions 82

Figure 52: Loss per training epoch for RNN on custom word2vec embeddings with lower case descriptions 83

Figure 53: Accuracy per training epoch for RNN on custom word2vec embeddings with lower case descriptions 83

Figure 54: Classification Report for RNN on custom word2vec embeddings with lower case descriptions 84

Figure 55: Loss per training epoch for RNN on custom FastText embeddings with lower case descriptions 85

Figure 56: Accuracy per training epoch for RNN on custom FastText embeddings with lower case descriptions 85

Figure 57: Classification Report for RNN on custom FastText embeddings with lower case descriptions 86
Figure 58: Model comparison with respect to metrics (weighted) 88
Figure 59: Testing accuracy per embedding type 89
Figure 60: Confusion Matrix of MNB on BoW and lower case descriptions 100
Figure 61: Confusion Matrix of MNB on BoW and lower case and stemmed descriptions 100
Figure 62: Confusion Matrix of DT on BoW and lower case descriptions 101
Figure 63: Confusion Matrix of DT on BoW and lower case and stemmed descriptions 101
Figure 64: Confusion Matrix of SVC on BoW and lower case descriptions............102
Figure 65: Confusion Matrix of SVC on BoW and lower case and stemmed descriptions 102
Figure 66: Confusion Matrix of FNN on BoW and lower case descriptions............103
Figure 67: Confusion Matrix of FNN on BoW and lower case and stemmed descriptions 103
Figure 68: Confusion Matrix of FNN trained on Google's word2vec (Glove) embeddings with lower case descriptions...103
Figure 69: Confusion Matrix of FNN trained on custom word2vec embeddings (with skipgram) with lower case descriptions...104
Figure 70: Confusion Matrix of FNN trained on custom FastText embeddings (with skipgram) with lower case descriptions...105
Figure 71: Confusion Matrix of CNN trained on Google's word2vec (Glove) embeddings with lower case descriptions...105
Figure 72: Confusion Matrix of CNN trained on custom word2vec embeddings (with skipgram) with lower case descriptions...106
Figure 73: Confusion Matrix of CNN trained on custom FastText embeddings (with skipgram) with lower case descriptions...106
Figure 74: Confusion Matrix of RNN trained on Google's word2vec (Glove) embeddings with lower case descriptions...107
Figure 75: Confusion Matrix of RNN trained on custom word2vec embeddings (with skipgram) with lower case descriptions...107

Figure 76: Confusion Matrix of RNN trained on custom FastText embeddings (with skipgram) with lower case descriptions 108

Contents

1 INTRODUCTION....16

2 TEXT CLASSIFICATION....18

2.1 EXISTING WORK ON THE PROBLEM....18

3 DATA REPRESENTATION AND ALGORITHMS....20

3.1 BAG OF WORDS (BOW) AND TERM FREQUENCY-INVERSE DOCUMENT FREQUENCY (TF-IDF)....20

I. MULTINOMIAL NAÏVE BAYES....22

II. DECISION TREES....24

III. SUPPORT VECTOR MACHINES....26

IV. FEEDFORWARD NEURAL NETWORKS....29

3.2 SEQUENCES OF WORDS AND WORD EMBEDDINGS....34

I. RECURRENT NEURAL NETWORKS....38

II. CONVOLUTIONAL NEURAL NETWORKS....42

4 DATASET....46

5 METHODOLOGY....48

5.1 BAG OF WORDS....48

5.2 SEQUENCES OF WORDS AND PRE-TRAINED EMBEDDINGS....49

5.3 SEQUENCES OF WORDS AND CUSTOM EMBEDDINGS....51

5.4 EVALUATION METRICS....52

6 RESULTS AND DISCUSSION 53

6.1 CLASSIFICATION RESULTS PER MODEL (ACCURACY, RECALL, PRECISION AND F1-SCORE) 53

6.2 MODEL COMPARISON 87

7 IMPLEMENTATION ISSUES 91

8 CONCLUSION AND FUTURE WORK 92

9 BIBLIOGRAPHY 94

APPENDIX 99

1 Introduction

The purpose of this Thesis is the evaluation of different Machine Learning and Deep Learning algorithms on the classification of job adverts to a general category. This work is motivated by the immediate need for application of such a tool in a real-world use case: Adzuna [1] is a search engine for job advertisements with millions of entries, which uses web scraping for collecting and processing of open job positions. As in many other search engines, Adzuna offers job seekers the opportunity to discover open job offerings by a general category (from a total of 29 categories), which can help narrow down searches to a viable amount with respect to the seeker's needs. Such tools – often called "browse by category" - are found very often in all kinds of search engines, making them vital for almost any web-based platform with advertisements. The used job categories are presented in Table 1. As millions of new ads get collected and displayed every day in the search engine, there is an emerging need for developing fast and efficient auto-categorization methodologies.

Category	id	Category	id
Accounting/finance	1	Logistics/Warehousing	16
Admin	2	Maintenance	17
Agriculture/Fishing/Forestry	3	Manufacturing	18
Consultancy	4	Other/General	19
Creative/Design	5	Public relations/Advertising/Marketing	20
Customer Services	6	Property	21
Domestic Help/Cleaning	7	Retail	22
Energy/Oil/Gas	8	Sales	23
Engineering	9	Scientific/Quality Assurance	24
Graduate	10	Security/Protective Services	25
Human Resources	11	Social Work	26
Healthcare/Nursing	12	Teaching	27
Hospitality/Catering	13	Trade/Construction	28
Information Technology	14	Travel	29
Legal	15		

Table 1: The job categories used by Adzuna, serving as labels in the classification task

Adzuna was using a rule-based approach where ads were categorized with respect to the source in which they were scraped. Every time a new source of ads got integrated within the scrapping framework, the developer would choose one of the categories for the whole source based on his personal beliefs about the proper category for the source. That said, there were multiple occasions where one source of 1000 ads for example would be marked as it-jobs simply because 600 of them were it related. This would mean that then rest 400 ads would be misclassified as it-jobs. The data science team of the company suggested that this system has an accuracy score of about 55% and that any machine learning system with a score of at least 75% would suffice as a replacement. Besides the low classification accuracy, the old implementation has also other bottlenecks. The first disadvantage is the bias that is introduced via the choice of the developer which assigns the category to the source of the ads. Secondly, it is highly impossible that a specific source contains ads of only one category. Even in the cases of sources that display domain specific ads such as the National Health System (NHS) of the UK, the distribution of real job categories is high. For instance, the NHS job site [2] (and therefore every ad scraped from there) would be mapped by the old implementation to the healthcare job category. This has led to major misclassifications, as the NHS job site contains also other types of offerings, such as ads seeking for sanitors and warehouse employees to work in the healthcare facilities. As a result, there was a need for an approach which could relate job description to the job categories with better generalization ability. To this end, this Thesis explores the capabilities of various Machine Learning and Deep Learning techniques in addressing the issue of classifying automatically new job advertisements into one of the pre-defined classes (job categories).

2 Text Classification

The methodology of the current work is subject to Text Classification, which is a field of study of Natural Language Processing. In general, Text Classification in Computer Science deals with the task of assigning a document to one or more classes algorithmically [3]: There is a set of entries $D = \{X1, \ldots, XN\}$ of dimension N, where each entry is labeled with a class withing the range of discrete class values indexed by $\{1, \ldots, k\}$ [4]. With respect to the mathematical representation of the input Data, the goal is to develop a classification model, which finds an estimate function which best describes the relation between the input feature representation and the classes. Depending on the classification algorithm that is used, the prediction may be accompanied by a probability (confidence measure), which indicates how strong the prediction is. In terms of Machine Learning principles, the input to the model training is set of hand labeled text documents along with the respective classes $\{(d_1,c_1)$, … , $(d_m,c_m)\}$ and the output is learned classifier $\{y: d \rightarrow c\}$.

2.1 Existing work on the problem

Text classification with Machine Learning has been widely researched in the last two decades. There are hundreds of applications including target marketing, medical diagnosis, news group filtering and document organization [3]. With respect to the purpose of the developed classifiers, the most used applications can be divided into two broader categories, which are topic classification and sentiment classification.

Topic classification is the task of text classification where the class to be predicted is a topic. It finds numerous applications such as news recommendation [5] and social media posts classification [6, 7]. It has also been applied in Social Science, where it was shown that the use of Machine Learning can help lower the costs of classifying large numbers of complex documents by 80% or more [8]. Additionally, text classifiers which predict the topic of a given document are also applied in the Legal Science for the classification of legal concepts [9]. Finally, Topic classification is found also in the automated email classification, where inbox emails are classified as per their content (e.g. offerings, travelling etc.) [10].

In sentiment classification, the goal is to predict the sentiment (label) of textual documents. It has been applied in social media in order to gain insights on how users think regarding a specific matter [6, 11, 12]. Via this, social media posts can be automatically classified to both the topic and then the sentiment (negative, neutral or positive). Variations of this application are also found: researchers have tried to train regression models in order to predict a continuous value within a range, which represents the sentiment. Such classifiers are also applied in movie reviews, in order to automatically extract conclusions on the viewers thoughts regarding specific movies [13, 14].

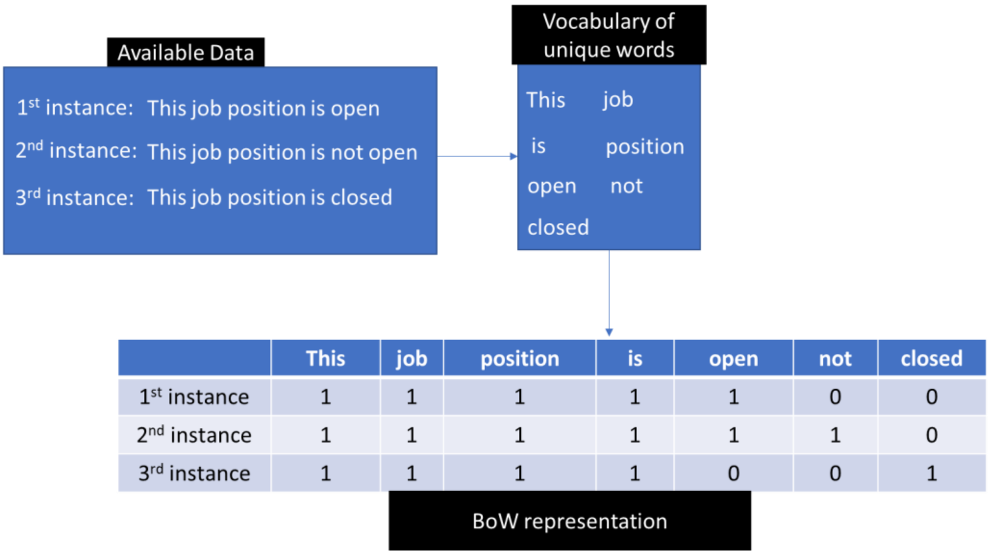
Finally, there are other categories of text classification such as spam detection, where the task is to classify inbox mail or messages as spam (or unwanted) [15, 16]. Unsupervised learning has also been applied for such purposes [17]. This has been further extended in social media, where developed classifiers are used to determine on whether a review comment is deceptive. Specifically, text classifiers are trained to detect synthesized reviews in product pages and identify spammers who try to promote some products or demote competitors’ products [18, 19]. User profiles of social media which make spam posts are also being identified with help of text classification techniques [20]. Text Classification techniques have also been applied for the identification of the author of online messages, taking into consideration various features of the writing style such as lexical, syntactic, structural, and content-specific features [21].

3 Data representation and algorithms

There are numerous conventional Machine Learning and Deep Learning algorithms that are used for text classification tasks. However, the algorithm that is used depends usually on the input data representation that is selected. Textual information needs to be converted into numbers, in order to be used as input to any algorithm. To that end, Text Classification includes an important first step: Transformation of the text data to an appropriate numerical representation. It is essential to define and describe the two data representations that are used in this work.

3.1 Bag of words (BoW) and Term Frequency-Inverse Document Frequency (TF-IDF)

BoW is a form of representing text in numbers. Essentially, BoW results in vectors for each textual instance. The first step of the BoW methodology is to identify all unique words that exist in the entire corpus of collected Data. Then, we can represent each entry in our Data with a vector of length equal to the number of unique words present in the entire corpus. Each point of the instance vector can take then the value 0 (does not exist) or 1 (exists) with respect to the existence of the respective word in the instance. Figure 1 below illustrates an example of BoW.


Available Data
1st instance: This job position is open
2nd instance: This job position is not open
3rd instance: This job position is closed
Vocabulary of unique words
This
job
is
position
open
not
closed
BoW representation

	This	job	position	is	open	not	closed
1st instance	1	1	1	1	1	0	0
2nd instance	1	1	1	1	1	1	0
3rd instance	1	1	1	1	0	0	1

Figure 1: An example of transforming text Data to BoW vectors

Term frequency–inverse document frequency (TF-IDF), is a numerical statistic that highlights the importance of a word to a document in a collection or corpus [22]. In order to calculate the TF-IDF value, we need first to calculate the Term Frequency (TF) and Inverse Document Frequency (IDF). TF is a measure of how frequently a term, t, appears in a document d and is calculated by the following equation:

$$tf_{t,d} = \frac{n_{t,d}}{Total\ number\ of\ terms\ in\ document}$$

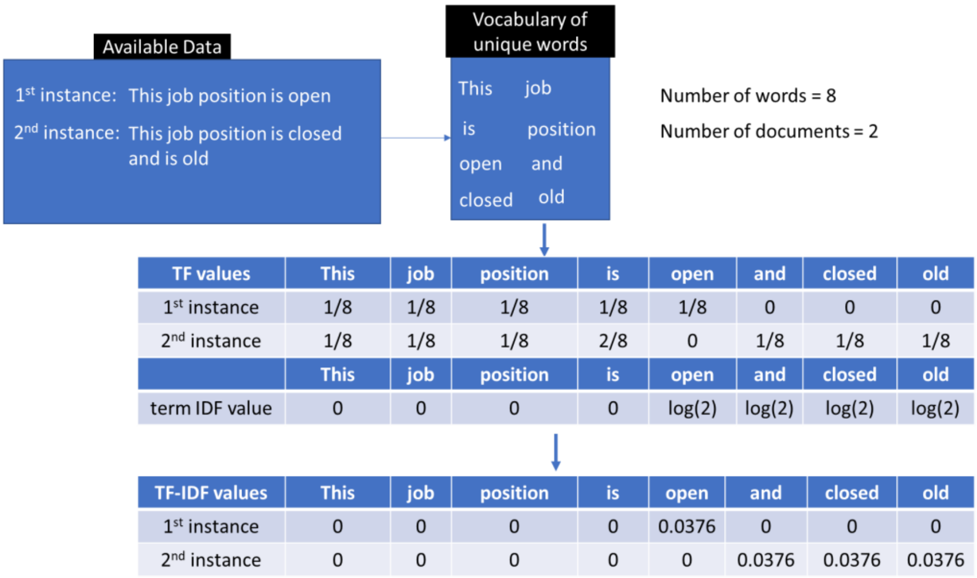
, where $n_{t,d}$ is the number of times that the term t appears in the corpus d. The IDF is used for introducing a value that reflects the importance of term. Specifically, it is computed by:

$$idf_t = \log\left(\frac{total\ number\ of\ documents}{total\ number\ of\ documents\ with\ term\ t}\right)$$

Finally, the TF-IDF of a term in a specific document is calculated by:

$$tf\_idf_{t,d} = tf_{t,d} * idf_t$$

The final feature representation of every instance is again a vector of length equal to the number of unique terms (words) in the entire corpus. Here, the value of each point in the vector is the $tf\_idf_{t,d}$ and not 0 or 1 as in simple BoW. Figure 2 illustrates an example.


Available Data
1st instance: This job position is open
2nd instance: This job position is closed and is old
Vocabulary of unique words
This
job
is
position
open
and
closed
old
Number of words = 8
Number of documents = 2

TF values	This	job	position	is	open	and	closed	old
1st instance	1/8	1/8	1/8	1/8	1/8	0	0	0
2nd instance	1/8	1/8	1/8	2/8	0	1/8	1/8	1/8
	This	job	position	is	open	and	closed	old
term IDF value	0	0	0	0	log(2)	log(2)	log(2)	log(2)

TF-IDF values	This	job	position	is	open	and	closed	old
1st instance	0	0	0	0	0.0376	0	0	0
2nd instance	0	0	0	0	0	0.0376	0.0376	0.0376

Figure 2: An example of TF-IDF vectors

I. Multinomial Naïve Bayes

The TF-IDF vectors are very often used as input data representation for feeding classification algorithms. Multinomial Naïve Bayes (MNB) is a classic algorithm in text classification. It originates from the Bayes Theorem, which describes the probability of an event to occur, based on prior knowledge of conditions relevant to the event [23]. It is described by the following mathematical formula which calculates conditional probabilities:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Where A and B are events and P(B) is not zero. $P(A|B)$ is the likelihood of event A to occur given that event B is true. $P(B|A)$ is the likelihood of event B to occur given that event A is true. Both $P(A|B)$ and $P(B|A)$ are also called a conditional probability. P(A) and (PB) are the probabilities of observing A and B respectively. They are also called marginal probabilities.

To apply Bayes' Rule to documents and classes, we transform introduce the class c and the document d to theorem as bellow [24]:

$$P(c|d) = \frac{P(d|c)P(c)}{P(d)}$$

Then, the most likely class (in a given prediction) will be:

$$c_{MAP} = argmax_{c \in C} P(c|d) = argmax_{c \in C} \frac{P(d|c)P(c)}{P(d)}$$

$$= argmax_{c \in C} P(d|c)P(c)$$

, where P(d) can be considered equal to 1. The subscript "MAP" in the class c on the left if the above equation refers to "maximum a posteriori" (most likely class). If we represent the document d as features (x_1, … , x_n), we get:

$$c_{MAP} = argmax_{c \in C} P(x_1, \dots, x_n|c)P(c)$$

The above equation is the fundamental principle of Multinomial Naïve Bayes Classifiers applied on BoW representations. It can be transformed to:

$$c_{NB} = argmax_{c_j \in C} P(c_j) \prod_{x \in X} P(x|c)$$

There are two assumptions of MNB applied on BoW representations. Firstly, the position does not matter; this is an assumption bound to the representation. Secondly, conditional independence is an MNB assumption: we assume that the feature probabilities $P(x_i|c_j)$ are independent given the class c, which means:

$$P(\mathrm{x}_1, \dots, \mathrm{x}_n|c) = P(x_1|c) * P(x_2|c) * \dots * P(x_n|c)$$

The first step of the algorithm is to extract the vocabulary of the entire dataset. Then we calculate the P(cj) terms for each class c_j in C (list of classes) as below:

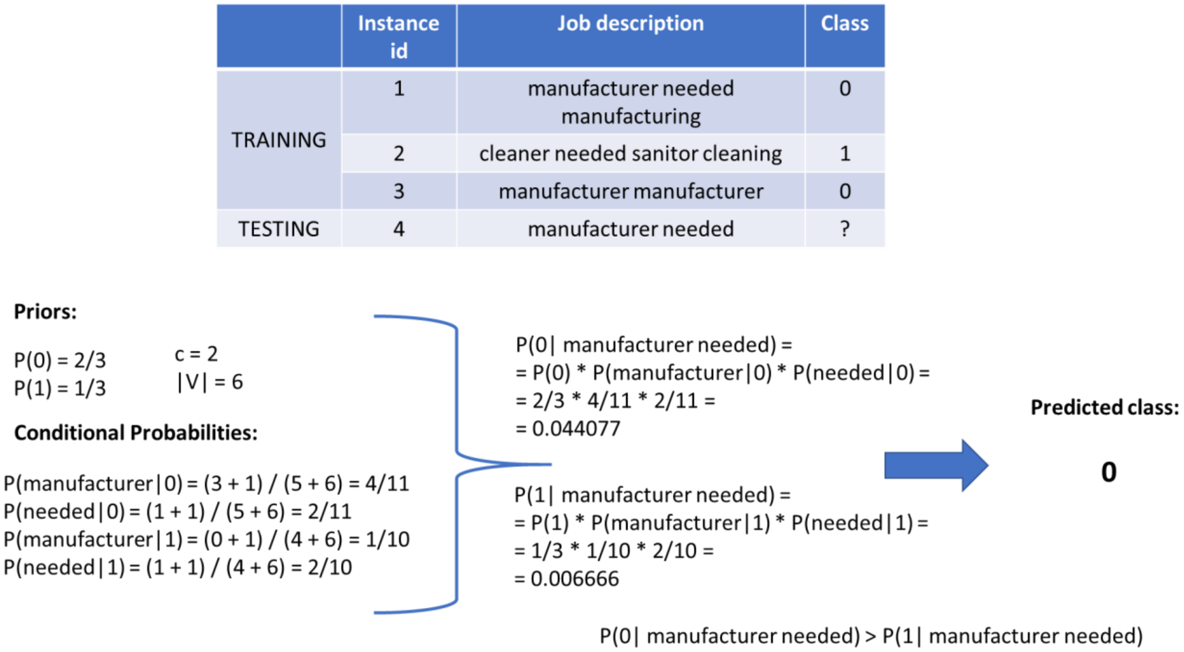
$$P(c_j) = \frac{amount\ of\ docs_j}{amount\ of\ total\ documents}$$

, where $docs_j$ are the documents with class j.

Then we calculate the $P(w_k|c_j)$ terms (fraction of times word w_k appears among all words in documents of class c_j) for each word w_k in the Vocabulary V:

$$P(w_k|\ c_j) = \frac{n_k+a}{n + a*len(V)}$$

, where n_k is the number of occurrences of w_k in $Corpus_j$ (single document containing all documents with class j). The term “a” is called alpha value for smoothing of the MNB algorithm and is introduced to discard the influence of words absent in the vocabulary. Setting a greater or equal to zero helps prevent zero probabilities in further computations [25]. In case where a equal to 1, it is called Laplace smoothing, while $\alpha<1$ is called Lidstone smoothing. Figure 3 illustrates an example of applying MNB to text documents (the BoW representation is skipped for better visualization).



	Instance id	Job description	Class
TRAINING	1	manufacturer needed manufacturing	0
	2	cleaner needed sanitor cleaning	1
	3	manufacturer manufacturer	0
TESTING	4	manufacturer needed	?

Priors:

P(0) = 2/3
P(1) = 1/3

c = 2
|V| = 6

Conditional Probabilities:

P(manufacturer|0) = (3 + 1) / (5 + 6) = 4/11
P(needed|0) = (1 + 1) / (5 + 6) = 2/11
P(manufacturer|1) = (0 + 1) / (4 + 6) = 1/10
P(needed|1) = (1 + 1) / (4 + 6) = 2/10

P(0| manufacturer needed) =
= P(0) * P(manufacturer|0) * P(needed|0) =
= 2/3 * 4/11 * 2/11 =
= 0.044077

P(1| manufacturer needed) =
= P(1) * P(manufacturer|1) * P(needed|1) =
= 1/3 * 1/10 * 2/10 =
= 0.006666

Predicted class:

0

P(0| manufacturer needed) > P(1| manufacturer needed)

Figure 3: MNB example on text data with alpha value equal to 1

Naïve Bayes classifiers have worked well in many real-world problems (multiclass classifications) and have a good reputation in the field of text classification (topic classification and spam filtering) [26, 27]. In addition, they usually require a small amount of training Data in order to achieve decent results. They are also fast and have low computational requirements. However, there are some disadvantages that need to be taken into consideration: Naïve Bayes assumes that all features are independent, which sometimes may not be realistic. In addition, NB algorithms assigns zero probability to features (words in our case) that have not been seen in the training data. When lots of Data are available, other more sophisticate algorithms are preferred such as Support Vector Classifier [24].

II. Decision Trees

Another popular algorithm which is used for text classification is the ID3 Algorithm, which belongs to the general category of Decision Trees (DT). DTs are non-parametric and are used for both classification and regression tasks. It is based on learning decision rules (multiple if nodes) from the input data. DTs consist of nodes, branches and leaves. In every node, there is a test for a feature value: there is one branch for every possible value that a feature can have [28]. All nodes (and branches) lead finally to a leave, which represents a specific class label. The objective is to develop multiple classification rules that can determine the output variable[29]. The classification of the instances is done by sorting them down to the developed tree from the root node all the way down to a leaf node, after testing all if statements on the path.

The methodology, with which the ranking (priority) of the feature value tests are assigned to the leaves, is called Information Gain. It is a statistical property and is used to determine the next attribute to split the decision tree on. Information Gain is calculated for each feature: features of high importance split the input data to sets that are relatively (to other features' splits) pure in one label. In other words, Information Gain measures how well a given feature separates the training instances with respect to the output variables [28]. Information Gain is calculated with the help of Entropy. Given a set of examples S, the entropy is calculated by:

$$Entropy(S) = \sum_{i=1}^{c} -p_i * log_2 p_i$$

, where p_i is the probability of S belonging to class i and c the set of all labels. Information Gain G(S,A) of an attribute A in a set of examples s can then be calculated by:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v),$$

, where Values(A) is a set with all observed values of the feature A. S_v is the subset of the entire S set, for which the attribute A has the value v. After calculating Information Gain for every available feature, we rank them as per their priority in the Decision Tree's node tests. Specifically, attributes with higher Information Gain are tested first. We split the set into subsets using the attribute for which the resulting information gain is maximized. Then, we create a node for this attribute: this node will have a number of branches equal to the number of observed values of the attribute in the training set. For each possible branch (feature value), we recurse with creating again subsets with the remaining set. The main advantages of using Decision Trees include the low computational cost of prediction (logarithmic in the number of instances used to train the tree) and the ability to handle both numerical and categorical data. In addition, DTs are easy to understand and interpret: in contrast to artificial neural networks and other more sophisticated algorithms, DTs decisions are easy to explain with Boolean logic. There also variations of DTs which can help address multi class outputs. On the other hand, are prone to over-fitting. In addition, DTs are also very sensitive: small changes in the training data may result into completely different trees. Ensemble methods are used for addressing such issues. Finally, DTs are usually biased towards dominant classes (in the training set), meaning that over or down sampling methods may be required before training. In this work, we experiment with the CART Decision Tree algorithm, which is based upon the C4.5 algorithm. C4.5 is the successor to ID3 and removed the restriction that features must be categorical by dynamically defining a discrete attribute (based on numerical variables) that partitions the continuous attribute value into a discrete set of intervals. C4.5 converts the trained trees into sets of if-then rules. The accuracy of each rule is then evaluated to determine the order in which they should be applied. Pruning is done by removing a rule’s precondition if the accuracy of the rule improves without it. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node.

III. Support Vector Machines

Another algorithm that is suitable to be trained on the BoW data representation are the Support Vector Machines (SVMs), which are used both for regression and classification. They have been used in topic classification and sentiment analysis [30, 31] and is considered to be effective in text classification [32]. SVMs have also been proven to perform well on imbalanced text datasets for text classification [33]. Finally, a comparison study between various conventional Machine Learning algorithms and their ability to discriminate the subtopics of Stock Exchange texts showed that SVMs along with Logistic Regression outperform all other algorithms when trained on TF-IDF data representations [34]. There are several reasons why SVMs outperform usually the rest of the traditional Machine Learning algorithms in text classification. These are the ability of the SVMs to handle high dimensional input space (and therefore text) in such a way, that no overfitting occurs [35]. Additionally, sometimes in text classification, only a few features are irrelevant for determining the class. A good performing classifier should use as many features as possible, which

SVM are proven to do [35]. As mentioned previously, resulted text vectors (BoW etc.) are sparse, as they usually contain only a few non zero values. SVMs have an inductive bias and are therefore highly appropriate in such cases [36]. Finally, many text classification tasks are linearly separable. Finding such a linear separator is the objective of SVMs with linear kernel [35].

The objective of SVMs is to find a hyperplane which best separates the instances into the classes (where best means minimum value of a given loss function). Depending on the kernel type (explained later), the hyperplane can be of different shape. The data points that are closest to the hyperplane are called Support Vectors. In the case of a 2-dimensional space and a linear kernel, this hyperplane is a line (linear function of one dimension) and the data points (2d vectors) with minimum distance to this line are the Support Vectors. In other words, any kernel type SVM tries to maximize the distance between the classes by learning one or more decision boundaries. The region which is defined between the Support Vectors and the line (or the hyperplane) is called the margin. Then, for a new previously not seen instance, the SVM assigns to it the class of the respective decision boundary it is in. An example of linear kernel SVM applied on a 2-dimensional space is illustrated in Figure 5 below.

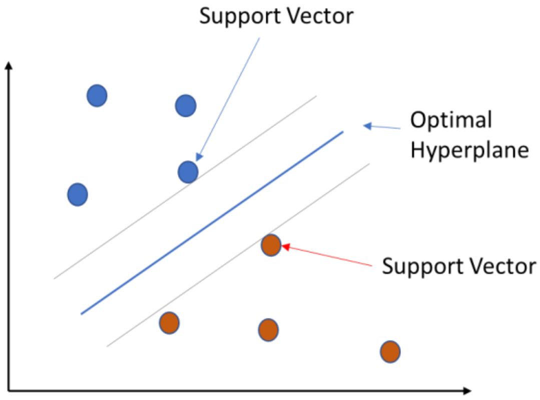


Figure 4: A simple example of 2d space and SVM hyperplane with linear kernel

For the case of the 2 dimensional space, SVM algorithm finds all lines that best classify (in terms of loss) the training data and finally choose the one, which has the greatest distance to the nearest data points (Support Vectors). In addition, SVMs have a parameter, called C, which allows the margin to get wider or narrower. Higher values of C take the margin region closer to the hyperplane. This can help classify correctly more training Data. On the other hand, choosing a lower C value allows for errors, thus being a good choice when overfitting needs to be addressed (Figure 6).

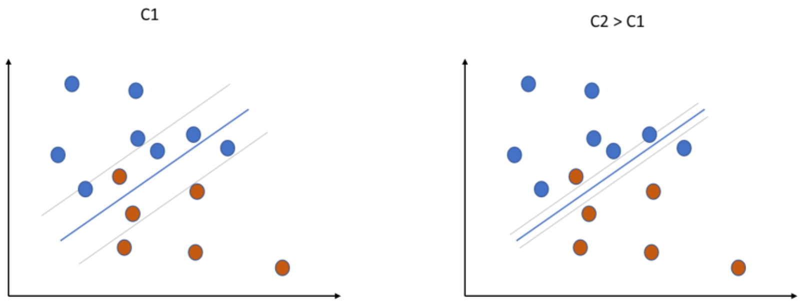


Figure 5: Comparison of margin when choosing different margins: Increasing the C parameters, the margin shrinks

The above description of the SVM algorithm refers to data that is linearly separable in the 2d case. In other cases, SVMs project the data to a space where they become linearly separable. The idea behind this is that by adding extra dimensions, there is a chance of getting linearly separable data. Projecting the data into higher dimensions is a job of the SVM kernels. Note that the described 2d case of above is using a linear kernel, which is the case where we do not do any projection to another space. Linear kernels simply compute the dot products in the original space. A kernel is a function which takes as input two points in the original space and computes the dot product in the projected space. Kernels essentially transform input data representation to the appropriate form with respect to the dimensional space we are investigating. Kernels can be linear, non-linear, polynomial, radial basis function (RBF) and sigmoid. The RBF kernel is defined as [37]:

$K(x_1,x_2) = \exp\left(\frac{||x_1 - x_2||^2}{2\sigma^2}\right)$, where σ is a free parameter. The polynomial kernel function is defined as:

$K(x_1,x_2) = ({x_1}^T x_2 + c)^d$, where d is the degree of the polynomial function we wish to choose, x_1 and x_2 the vectors from the original input space. “c” is a free parameter. The linear kernel is a polynomial kernel function with d =1:

$$K(x_1,x_2) = ({x_1}^T x_2 + c)$$

Finally, the sigmoid kernel is defined as:

$K(x_1,x_2) = tanh(ax_1{}^T x_2 + c)$, where both a and c are free parameters [38].

IV. Feedforward Neural Networks

The last algorithm used for the BoW data representation are Artificial neural networks (ANNs). ANNs are named after biological neural network of animal brains: they consist of nodes (called neurons) in which numbers (equivalent to signals in brain) come into and are transformed to an output via a non-linear function of the sum of input numbers. Nodes communicate with each other, just like neurons in the brain do. In the case of ANNs, nodes are connected with edges, which have weights that adjust while being trained on Data. The weight of each edge can be considered as a threshold above which the signal passes from one neuron to another; the next neuron will be activated. Essentially, ANNs have multiple layers of neurons. Different transformation (non-linear functions) may be present in each layer. Neurons of one layer communicate with the neurons of the next layer. This is called a feedforward network (FNN): Information passes in one direction from the first layer to the last one [39]. The first layer is called input layer, the last layer is called output layer and every layer in between belongs to the hidden layers. There are also other types of ANNs in which information can persist by forming cycles from different layers: such ANNs are the recurrent networks.

For the multi-layer feedforward neural networks, each neuron is connected to the neurons of the next layer. The learning process is based on the back-propagation algorithm, in which the output values are compared each time with the true value (expected) and the loss (error) is calculated. Subsequently, the calculated error is fed back (back-propagation) in the opposite direction: via this, the weights at each edge are adjusted in such a way that the error is reduced. Weights are adjusted with help of the non-linear optimization methods such as Gradient Descent: The derivative of the loss function with respect to the weights is calculated in order to find the direction (positive or negative) of change that needs to be made in the respective weight in order to reduce the error [40]. While fitting the training Data to the network, the backpropagation algorithm calculates the gradient of the chosen loss function with respect to each weight by the chain rule, which is by computing the gradient one layer at a time, iterating backward from the last layer [41]. Finally, there is Stochastic Gradient Descent (SGD), which is an algorithm for learning (new weights) using the gradient calculated by back-propagation [41]: SGD takes a mini-batch of data, updates the weights (including biases) based on the average gradient from the batch (gradients of the batch calculated by back-propagation). After repeating this process (feed-forward-ing the inputs and backpropagating the error), the network reaches a point where the error is not being reduced any further while adjusting the weights: this is called converging.

Feedforward Neural Networks are usually organized in layers: the input layer, some hidden layers and an output layer. Absence of hidden layers defines a single-layer perceptron, whereas one or more hidden layers define a Multilayer Perceptron. Figure XX illustrates a feedforward neural network architecture with m hidden layers. The input space is d-dimensional vector $\vec{x} = \{ x_1, \dots , x_d \}$. Here, we have n-dimensional hidden layers: every hidden layer has exactly n neurons. There are no connections between neurons of the same layer. In the neurons of the network (x_1, x_d , $s_{1,1}$ etc.), a mathematical transformation is taking place by a function called activation function. Every neuron in layer l $s_{l,j}$ is connected to every neuron in layer l-1. Every neuron depends on the output of all the neurons in the previous layer. In each of such connections, there are weights and bias. For instance, $s_{1,1}$ is connected with x_1 with $w_{x1,s11}$ and $b_{x1,s11}$. In each connection, the input value is multiplied by the weight and then the bias is added. This is denoted as h_j^k, with j being the neuron and k the layer.

This product is then given input to the activation function g. There are various activation functions which are used serving as gates in between input to a neuron and its output to the next layer. Activation functions can be simple enough, such as the binary step function, which are threshold-based, in which if the input value is above (or below) a certain threshold, the current neuron is activated and the exact same signal is sent to the next layer. In the case of multiclass classification, this cannot be used as output layer activation function, as it cannot support classifying to one of the many categories. Additionally, there are linear activation functions: they take the input and create an output signal which is proportional (y=bx) to the input. They allow for multiple output (not just binary output like the binary step functions) but they have major bottlenecks when coming to training: It is not possible to use them with the backpropagation algorithm, as they result in constant derivatives of the error function. Finally, there are non-linear activation functions. They are appropriate for backpropagating the error and identifying the updates needed for the weights and biases. Sigmoid function is one of the non-linear activation functions. It is defined as $S(x) = \frac{e^x}{e^x+1}$. It can be used also for the output layer. Output values range between 0 and 1 which also is a way of normalizing the output of each neuron. A great advantage of using sigmoid activation function relies on the fact that for high positive or negative values (-2<x<2), the output value is close to 0 or 1, which enables clear predictions. On the other hand, sigmoid activation function can result in the phenomenon called vanishing gradient, where for very high or very low values of input, there is no change to the prediction. This can lead to vanishing gradient (just small changes to the error), meaning the network does not learn any further or learn slowly. In addition, sigmoid functions can be computationally expensive. Last but not least, sigmoid functions do not offer zero centered outputs, which can be problematic in some tasks. When in need of zero centered output, one can use the hyperbolic tangent activation function, which is a good choice when there are polarized values (strongly negative, neutral and positive) in the input. Hyperbolic tangent activation function is defined as: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. Tanh can also be computationally expensive. On of the most common activation functions used is the rectified linear unit (ReLu) activation function. It is more efficient computationally compared to sigmoid and tanh and often helps the network converge quickly. It is defined as: $R(x) = \max(0, x)$. Although it allows for backpropagation, it can sometimes prevent the network from learning further: when inputs are zero or negative, the gradient of the function becomes zero and backpropagation cannot be performed. This can be avoided with use of a variation of ReLu called leaky ReLu, which has a small positive slope in the negative values, such to enable backpropagation. Leaky ReLu is defined as $R(x) = \max(0.01 * x, x)$. Finally, softmax activation function is another non-liner activation function, which is commonly used for training neural networks. It is able to handle multiclass problems. It also gives a probabilistic output (probability of the input value being a class). This is why it is often used only as output layer activation function in networks which learn to predict multiple classes. It is defined as $\sigma(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$ for j = 1, …, K, where x is the input vector of K real numbers.

Figure 7 illustrates the activation functions described here.

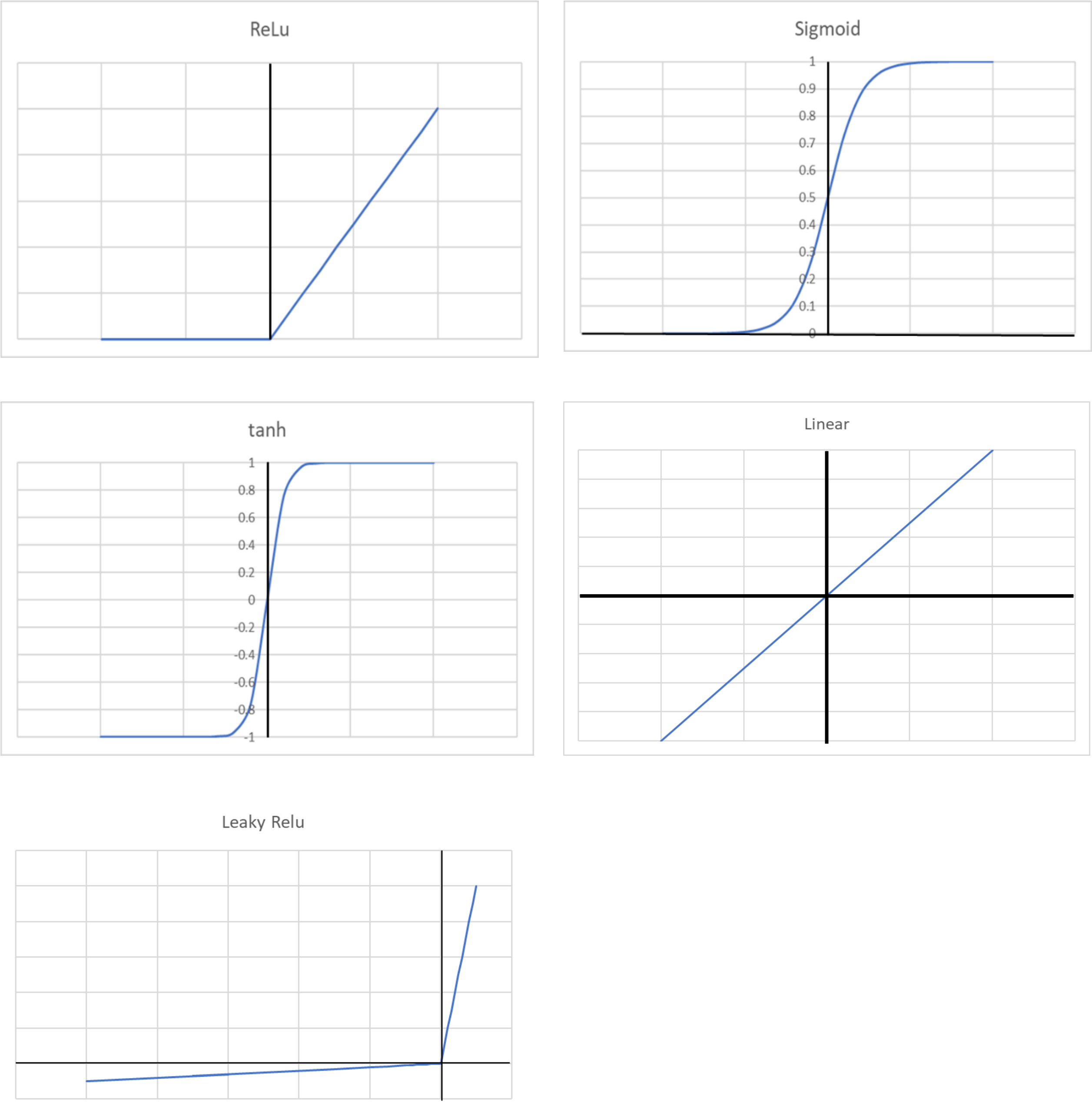


Figure 6: Activation functions

The output for this node is denoted as o_j^k. If we define as r_k the number of nodes in layer k, we can denote weights of a specific neuron j in layer k are highlighted as vectors: $\overrightarrow{w_j^k} = \{ w_{1j}^k, \dots, w_{r_k j}^k \}$. Respectively, the output vector for layer k can be denoted as vector $\overrightarrow{o^k} = \{ o_1^k, \dots, o_{r_k}^k \}$. In the beginning of the training, the input layer is initialized with output values o_i^0 which are equal to the input vector $\vec{x} = \{ x_1, \dots, x_d \}$, i.e. $o_1^0 = x_1$.

Then, the product sums and output of each hidden layer are calculated as described next. For every hidden layer (1 to m) k, we compute $h_i^k = \overrightarrow{w_i^k} \cdot \overrightarrow{o^{k-1}} + b_i^k$, where i takes all values between 1 and r_k (number of neurons in layer k). The h_i^k is fed to the activation function g and the output is calculated: $o_i^k = g\left(h_i^k\right)$ for i in range 1 and r_k. The activation function g can differ from layer to layer. Depending on the output type we need, we can also choose different activation functions in the output layer and the hidden layers. In any case, the output of the neural network will be: $o = g\left(h_{r_m}^m\right)$.

During training, a feedforward network tries to learn by updating the values of the weights $\overrightarrow{w_i^k}$ and the biases b_i^k. Given a set of training examples $X = \{ (\vec{x}_1, y_1), \dots, (\vec{x}_N, y_N)\}$, each time these parameters are updated, the mean squared error is calculated:

$MSE(X) = \frac{1}{2N} \sum_{i=1}^{N} (o_i - y_i)^2$, where o_i signifies the prediction made by the network regarding the input vector $\overrightarrow{x_i}$ and y_i the true value. The goal is to minimize the MSE(X). In each of such updates (iteration), the weights and biases are calculated with help of the gradient descent:

$$\Delta w_{ij}^k = -a \frac{\partial MSE(X)}{\partial w_{ij}^k} \text{ and } \Delta b_i^k = -a \frac{\partial MSE(X)}{\partial b_i^k}$$

The partial derivatives of the above equations are computed with help of backpropagation, which is an algorithm for calculating the error function (in this case MSE) with respect to the network's weights and biases. One single iteration of the network includes the following steps: First, all $o_i^k \; and \; h_i^k$ values are calculated for every input vector $\vec{x}_i$. Then, the backward values are calculated with help of backpropagating the error. Specifically, the partial derivatives of the error function are computed. Then, according to these derivatives, the weights and biases ($\overrightarrow{w_i^k}$ and b_i^k) are updated.

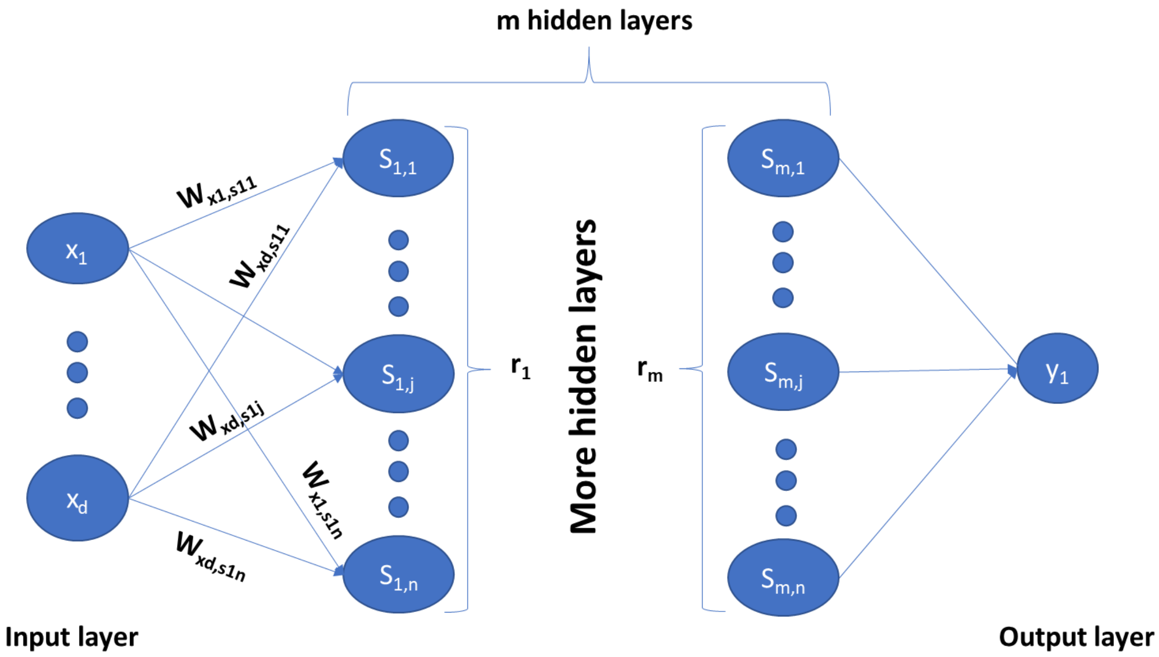


Figure 7: An example architecture of a feedforward multilayer perceptron.

3.2 Sequences of words and word embeddings

Another popular input representation of textual Data are word embeddings, which have been widely used in Text classification [42-44]. However, due their dimensions the use of more sophisticated Neural Networks such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) is preferred [45-47]. In order to transform textual data to the appropriate format, the text is represented as a multidimensional vector. Each of its dimensions are a one-dimensional vector which represents a word. These dense vectors are called word embeddings. Word or sequences of words from the available vocabulary of a task are mapped to vectors, which are a learned representation for text where words that have the same meaning also have a similar representation. Word2Vec, FastText and Glove are some of the most popular word embeddings that are used in NLP tasks.

Word2Vec representation is learned by the word2vec algorithm, which is an unsupervised algorithm which detects synonymous words and learns associations between words in a given corpus. In this input representation, each unique word (or lemma) is represented as a one-dimensional vector of numbers. The cosine similarity metric between two vectors A and B gives the level of semantic similarity between the respective words:

$$similarity = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Obtaining a Word2Vec representation can be done with two ways: using the continuous bag-of -words (CBOW) and the skip-gram model. Given a corpus, the model of word2vec loops on the words of each sentence and tries to use the current word in order to predict the neighboring words (context). If we can represent our text as a sequence of words:

$$w_o, w_1, \dots, w_{N-1}, w_N$$

Then, for word w_j (called central), the context of w_j is given by its left and right neighboring words:

$$w_{j-M}, \dots, w_{j-1}, w_j, \dots, w_{j+1}, w_{j+M}$$

where M is a selected parameter and represents the half-size of the context window, which is the number of words in each context. To each word w_i, a vector representation v is assigned. Also, we define the probability that w_o is in the context of w_i as the softmax of their vector product:

$$P(w_o|w_i) = \frac{e^{(v_{w_i} \cdot v_{w_o}^T)}}{\sum_{j=1}^{V} e^{(v_{w_i} \cdot v_{w_j}^T)}}$$

Skip-gram models try to predict the context of central words, by finding the set of vectors v which minimize the loss function:

$Loss\ Function = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j\in C_i}\log\left(P\left(w_j\middle|w_i\right)\right)$ where C_i is the context of word w_i.

Skip-gram model is a Neural Network, which has as input layer a vector of length X, where X is the number of unique words in the corpus. Each word has its own vector, with initially a "1" value in the position of the same word and zeros in every other position. The output vector has also length X and contains for every unique word in the corpus the probability that a randomly selected nearby word is the same word. The output layer uses the softmax described above. If two different words have very similar contexts, then the model will output similar vector results. Hidden layer units do not have activation functions. An example architecture of skip-gram modelling is illustrated in Figure 9. Finally, besides of the context window size (M) described above, there are also some other parameters that are important when training a word2vec representation with skip-gram model. These include both word2vec specific parameters such as the length of the representation vector as well as common neural network parameters such as the learning rate and the number of training epochs. Figure XX illustrates an example of word2vec representation. Note, that stopwords such as "is" and "this" are usually excluded from training and representing text as rarely capture any relevant information. Finally, we need to mention a major bottleneck of Word2Vec representations: the Out-Of-Vocabulary issue. As one embedding is created for each word, new words not seen during skipgram (or CBOW) training cannot be handled and are therefore assigned a zero value.

GloVe (Global Vectors for Word Representation) is a method for creating word2vec-like representations. It origins from the Stanford University and has various available open-source pre-trained embeddings such as the GloVe 100d (word embeddings of length 100) [48]. Glove model is trained on a matrix, which consists of word-word co-occurrence non zero values. This matrix essentially highlights how frequently word co-occur with other words within a corpus. GloVe tries to capture both global and local statistics of the given corpus [49].

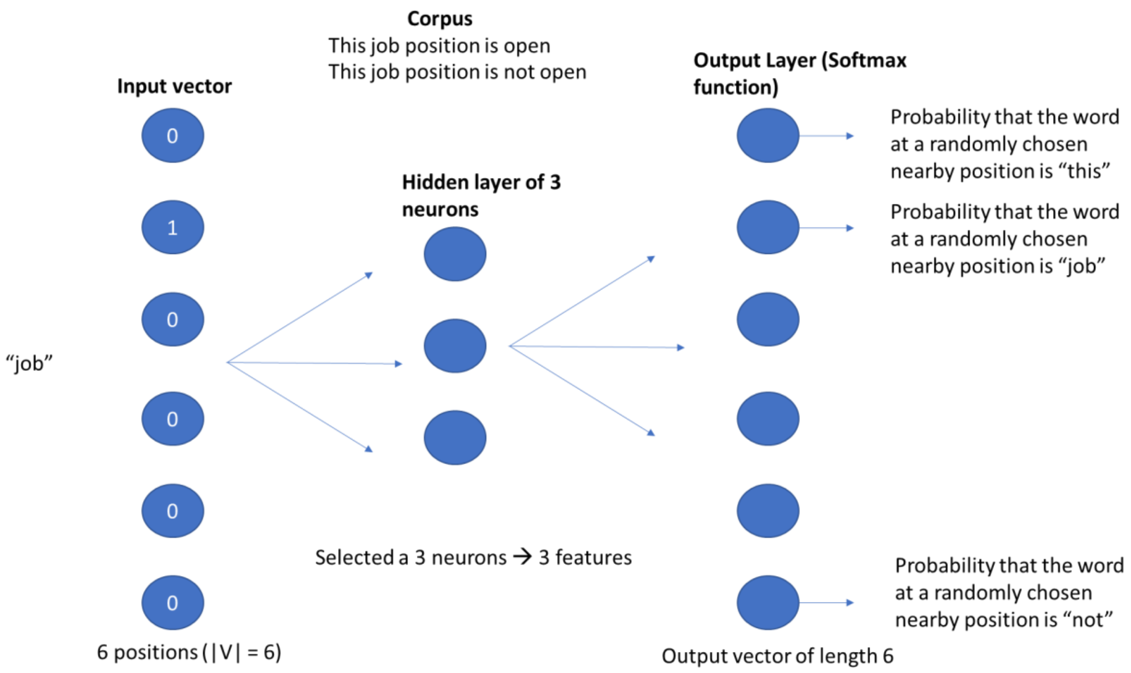

Corpus
This job position is open
This job position is not open
Input vector
0
1
0
0
0
0
"job"
6 positions (|V| = 6)
Hidden layer of 3 neurons
Selected a 3 neurons → 3 features
Output Layer (Softmax function)
Probability that the word at a randomly chosen nearby position is "this"
Probability that the word at a randomly chosen nearby position is "job"
Probability that the word at a randomly chosen nearby position is "not"
Output vector of length 6

Figure 8: Example architecture of a skip-gram training for word2vec embedding

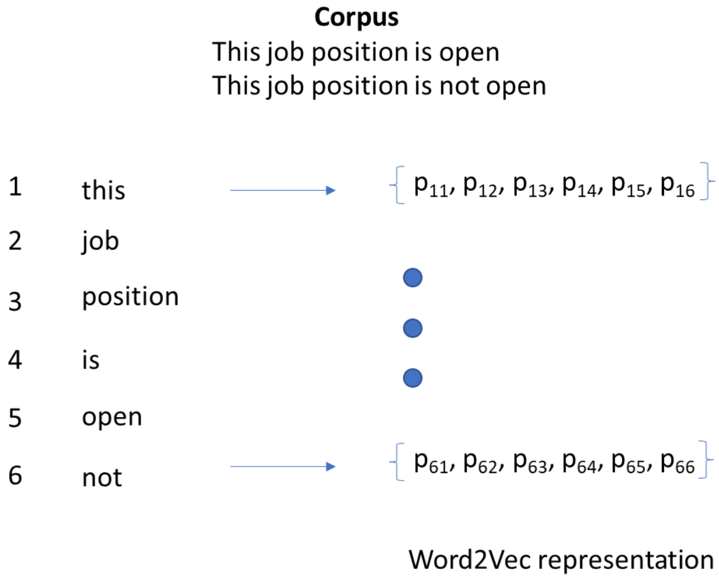

Corpus
This job position is open
This job position is not open
1 this
2 job
3 position
4 is
5 open
6 not
{ p_{11}, p_{12}, p_{13}, p_{14}, p_{15}, p_{16} }
{ p_{61}, p_{62}, p_{63}, p_{64}, p_{65}, p_{66} }
Word2Vec representation

Figure 9: Example of word2vec representation

Another popular technique for learning complex representations of text with temporal information is FastText. FastText was built upon Word2Vec and is based on the fact, that each word is not treated as a whole (as in Word2Vec) but as a composition of multiple ngrams. For instance, if we choose n equal to 3 the word "manufacturing" is treated as a vector of all 3-grams:

$$[< ma, man, anu, nuf, ufa, fac, act, ctu, tur, uri, rin, ing, ng >]$$

The special characters < and > are introduced in FastText at the beginning and end of original words. FastText representations can help overcome common Word2Vec issues such as a word being not in the vocabulary. FastText representations (embeddings) can be trained with skip-gram and CBOW [44]. Finally, Gensim Library provides open-source software for training both Word2Vec and FastText embeddings [50].

I. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a category of Deep Neural Networks which can help capture temporal information in the text, i.e. relationships between words and their position in the job advertisement. Long Short-Term Memory (LSTM) networks are a special case of RNNs. RNNs have been used for text classification tasks such as context sensitive email classification and web site indexing [51]. In addition, LSTMs have outperformed other methods achieving 87% classification accuracy in the binary task of classifying social media messages to one of the two United States political parties' leanings [46]. Sentiment Classification of user reviews is another problem which LSTMs have addressed efficiently [52, 53]. As mentioned before, RNNs are a general category of deep learning architecture. For the purposes of this work, we investigate specifically the LSTM architecture.

LSTMs are very popular for text classification tasks. They were developed explicitly to capture long-term dependencies and as a need to address the vanishing gradient problem: deep networks with multiple layers suffer from zero valued gradients of the loss function due to the activation functions applied, which leads to challenges in training (during backpropagation) [54]. In other words, the gradients decrease (or increase) exponentially while being multiplied consecutively in great numbers of layers. LSTMs use a technique called "gradient clipping" which deals with vanishing gradient problem by capping the maximum value for the gradient. LSTMs can learn long-term dependencies between input and output due to some architectural units called forget (or recurrent in general) gates. As in any RNN, LSTMs have loops in their units. An example of a typical RNN architecture is illustrated in Figure 11 below. All RNNs consist of repeating modules (units), which are presented in Figure 12. In standard RNNs, these modules have a similar and simple structure. Input x_t is feeded in a unit A and the output is h_t. In contrast to Feedforward Networks, there are loops in some (or all) units, allowing previous output to be used as inputs while having hidden states. For any timestep t, the activation (or input to unit) a_t and output h_t are defined as:

$a_t = g_1(W_{aa}a_{t-1} + W_{ax}x_t + b_a)$ and $h_t = g_2(W_{ha}a_t + b_h)$, where g_1 and g_2 are some activation functions and W_{ax}, W_{aa}, W_{ha}, b_a, b_h are coefficients that are shared temporally. Commonly used activation function are the sigmoid, the tanh and the ReLU functions.

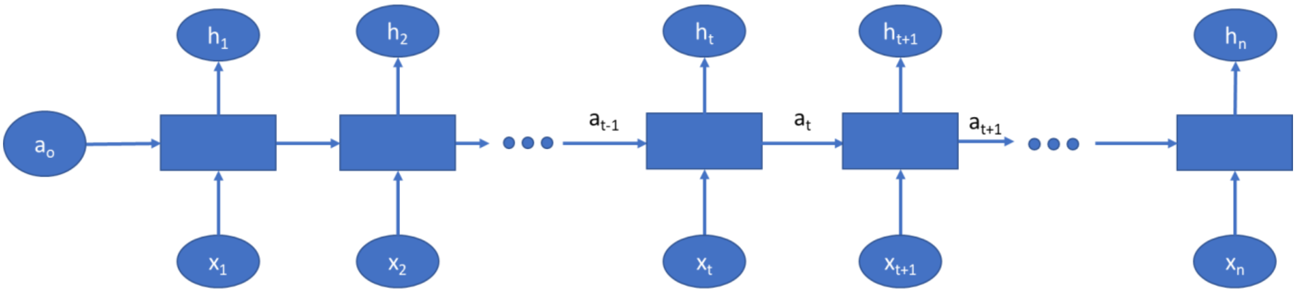


Figure 10: An example of recurrent architecture

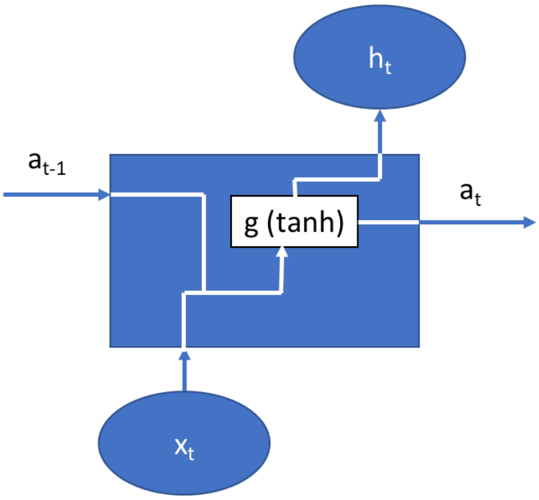


Figure 11: A simple recurrent unit in standard RNNs

The total loss function (all time steps) is calculated via all losses of every time step t:

$$L(h_{pred}, h_{true}) = \sum_{t=1}^{T_h} L(h_{pred}^t, h_{true}^t)$$

Backpropagating occurs also at every time step T:

$$\frac{\partial L_T}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_T}{\partial W}\bigg|_t$$

LSTMs have also a chain-like structure regarding the unit layout. However, the units have different structure themselves. In every unit (called cells or memory blocks), there are two states that are transferred to the next cell: these are the cell state and the hidden state (C_t and h_t). The cell state run straight through the entire cell, where three operational units called gates are adding or deleting information to it. Gates (Figure 13) consist of a sigmoid neural network layer and a pointwise multiplication operation. This layer output a value between zero and one (like any sigmoid function). This output value describes how much of the input should be passed through.

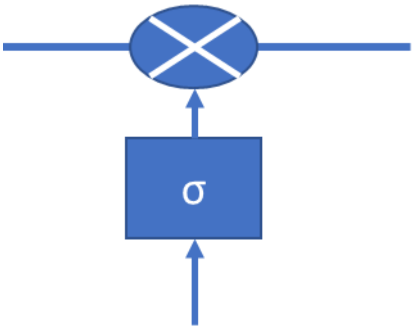


Figure 12: LSTM gate: A sigmoid neural network layer and a pointwise multiplication operation

Firstly, h_{t-1} hidden state from previous cell is coming in the cell. Through a sigmoid layer called “forget gate layer”, the network decides on what information will be thrown away from the cell state. This is done by “looking” at h_{t-1} and x_t and output a number between zero and one, where a zero means getting completely rid of the previous hidden state and a one signifies keeping the hidden state as is. The forget gate layer is a function defined as:

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

Subsequently, the network decides what new information is going to be stored in the cell state. This step consists of two sequential operations. At first, a sigmoid layer i_t (called input gate layer) decides which values will be updated. Afterwards, a tanh layer computes the candidate values $\tilde{C}_t$ that could be added to the cell state. These two operations are defined as:

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

Next, the old cell state C_{t-1} is updated and “integrated” to the new cell state C_t based on the operations done so far. Specifically, the old state is multiplied first by f_t, which results in “forgetting” the information that f_t function decided to forget. Next, the product of i_t and $\tilde{C}_t$ is added to the cell state C_t. The addition of $(i_t * \tilde{C}_t)$ signifies the new candidate values $(\tilde{C}_t)$ scaled by how much (i_t) the network decided to update the cell state. C_t is given thus by:

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

The final step is to compute an output hidden state h_t which will be used in the next cell (if any) along with C_t. First, a sigmoid layer decides which parts of the cell state are going to be out-put (o_t). A tanh function squishes the cell state values between -1 and 1 (tanh(C_t)). The outputs of these two operations are multiplied in order to apply the earlier decision on what parts to ouput and the final h_t is calculated:

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

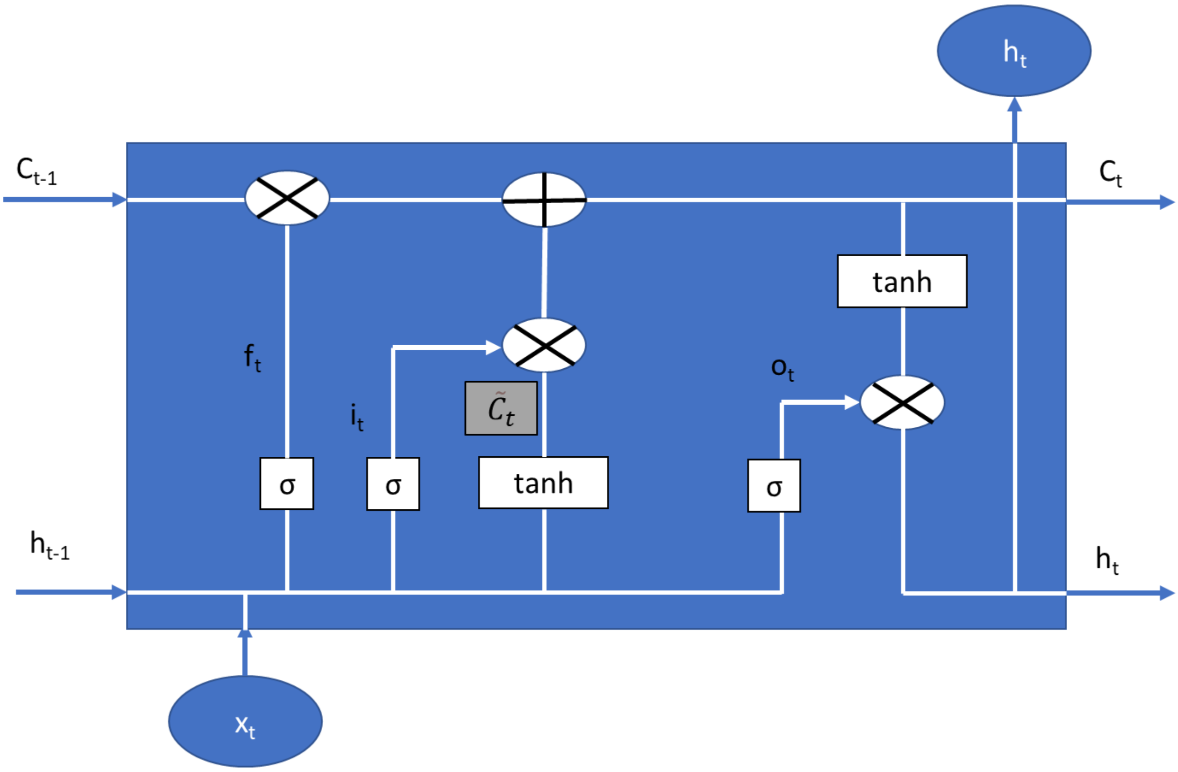
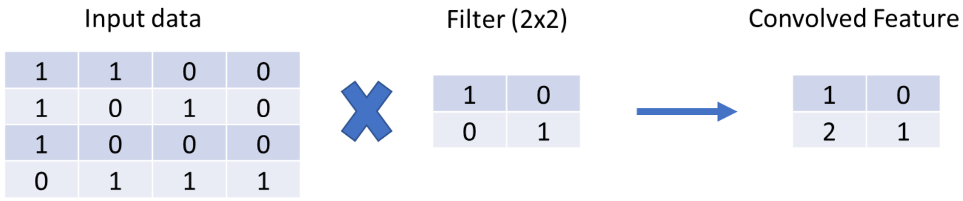


Figure 13: A LSTM cell

II. Convolutional Neural Networks

Convolutional neural networks (CNNs) have been used for text classification on word sequences input representation, showing good classification results compared to Long Short Term Memory models [3] and other conventional Machine Learning approaches [55]. Compared to the previous input representation (BoW) described, CNNs have been proved to outperform them on word sequences with at least 15% in terms of classification accuracy [47]. The depth of the model seems to be of high importance for better results in the task of classifying text with CNNs: very deep CNNs have outperformed existing state-of-the-art results [56]. Finally, CNNs have been combined with LSTMs for representing input sentences and classifying them [57, 58]. This approach was able to capture both important local (low level) features as well as global and temporal relationships with CNNs and LSTMs respectively achieving state of the art results in some real-world use cases [55, 57]. The main results of applying LSTMs (and RNNs in general) together with CNNs for the task of text classification show that CNNs can be very efficient at extracting features that are irrelevant with the position of the words, whereas RNNs and LSTMs can model sequential relationships [59].

CNNs consist of an input layer, one output layer and one or more hidden layers. However, CNNs differ essentially from feedforward multilayer perceptrons, as their layer are convolutional: they convolve with a multiplication or some other dot product. The convolutional operation occurs in the first part of the convolutional layer and is carried out by the Kernel (or filter), which is a matrix. The kernel begins on the top left of the input matrix, performs the convolution on the same length sub-matrix over which it is hovering and moves on to the right with respect to a certain stride value. When it parses the width of the input matrix, it continues from the next row of the input matrix. This continues until the entire input matrix is parsed. Kernels transform the input data to some other shape which captures neighboring relationships. Therefore, CNNs can capture the spatial or/and temporal dependencies in input data by applying filters. Usually, the first layers of convolution capture low level dependencies whereas the last convolutional layers capture more high level representations. These multiple convolution result in reducing the dimensionality of the input data. To avoid shrinking output due to loss of information, CNNs can be equipped with padding layers, which adds extra values on the input matrix (usually zeros), so as to result in same length convolved features (same padding). Typically, after having the convolutional filter applied on the input matrix, an activation function such as ReLU is passed over the convolved feature.



Input data

1	1	0	0
1	0	1	0
1	0	0	0
0	1	1	1

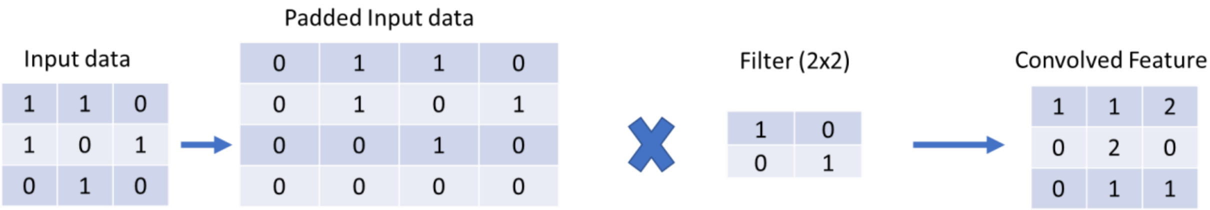
Filter (2x2)

1	0
0	1

Convolved Feature

1	0
2	1

Figure 14: An example of convolution with 2x2 filter on 4x4 input and stride length 2 on both axes



Input data

1	1	0
1	0	1
0	1	0

Padded Input data

0	1	1	0
0	1	0	1
0	0	1	0
0	0	0	0

Filter (2x2)

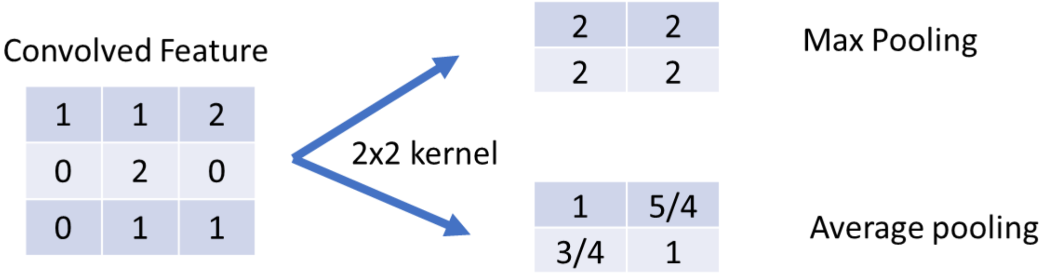
1	0
0	1

Convolved Feature

1	1	2
0	2	0
0	1	1

Figure 15: An example of convolution with 2x2 filter on 3x3 input and stride length 1 on both axes with zero padding

CNNs also have Pooling layers which reduce the dimensions of the convolved feature in order to extract further dominant features and to reduce the computational needs. We can discriminate two types of pooling functions. Max Pooling returns the minimum from the sub matrix covered by the convolutional kernel. Max Pooling also helps towards removing noise as it discards noisy activation by selecting always the maximum value of the neighborhood. Average Pooling returns the average of all values inside the sub matrix covered by the convolutional kernel. Average Pooling can also help regarding noise reduction.



Convolved Feature

1	1	2
0	2	0
0	1	1

Max Pooling

2	2
2	2

Average pooling

1	5/4
3/4	1

Figure 16: Max and Average Pooling over 3x3 convolved feature with 2x2 kernel

To sum up, every hidden layer of a CNN consists of the convolutional operation along with the pooling layer. The final output is given as input to a regular feedforward neural network in order to proceed with the classification. Before this, the final output needs to be flattened out, as after the convolutional layers it is represented as matrix. Therefore, an MxN matrix output of CNN convolutional layers is flattened out through a flattening layer into MxN vector {x_1, …, x_{MN}}. This vector is then fed to a feedforward ANN. There, every step described regarding the ANNs it taking place (gradient descent, backpropagation etc.). Finally, during training, CNNs try to learn filter values, which is equivalent to learning weights and biases in ANNs. CNN neurons can share the same filter, which makes CNNs much less computationally expensive [60]. Applied on text classification and specifically the sequence of words (with embeddings) input representation, the CNN has an equivalent architecture like Figure XX below: An MxN pre-trained word embeddings representation of the text is the input matrix. Multiple convolutional layers (convolution, ReLu) and pooling layers are following. The output is flattened and given as input to a feedforward multilayer perceptron, which has a softmax function in the output layer for classifying to one of the 29 categories.

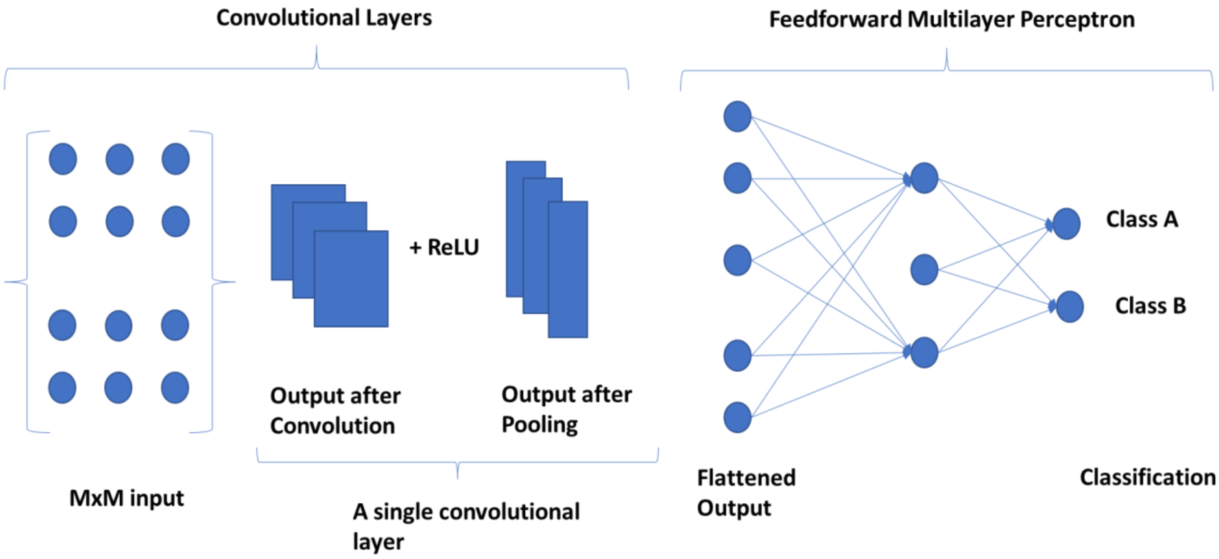


Figure 17: CNN architecture example

4 Dataset

In order to train the described algorithms, we collected 233.997 job descriptions of real-world job advertisements from the “Find a Job” application of the UK government [61]. Adzuna is the official provider and maintainer of all Data. These Data are free text job descriptions of length between 8 and 676 words that have been scraped by Adzuna. The labelling of the training Data is handcrafted: Every job description is manually annotated with one of the 29 categories (Table 1) that are used. For the evaluation of results, it is necessary to highlight the class distribution in the training, validation and testing dataset (Figure 19).

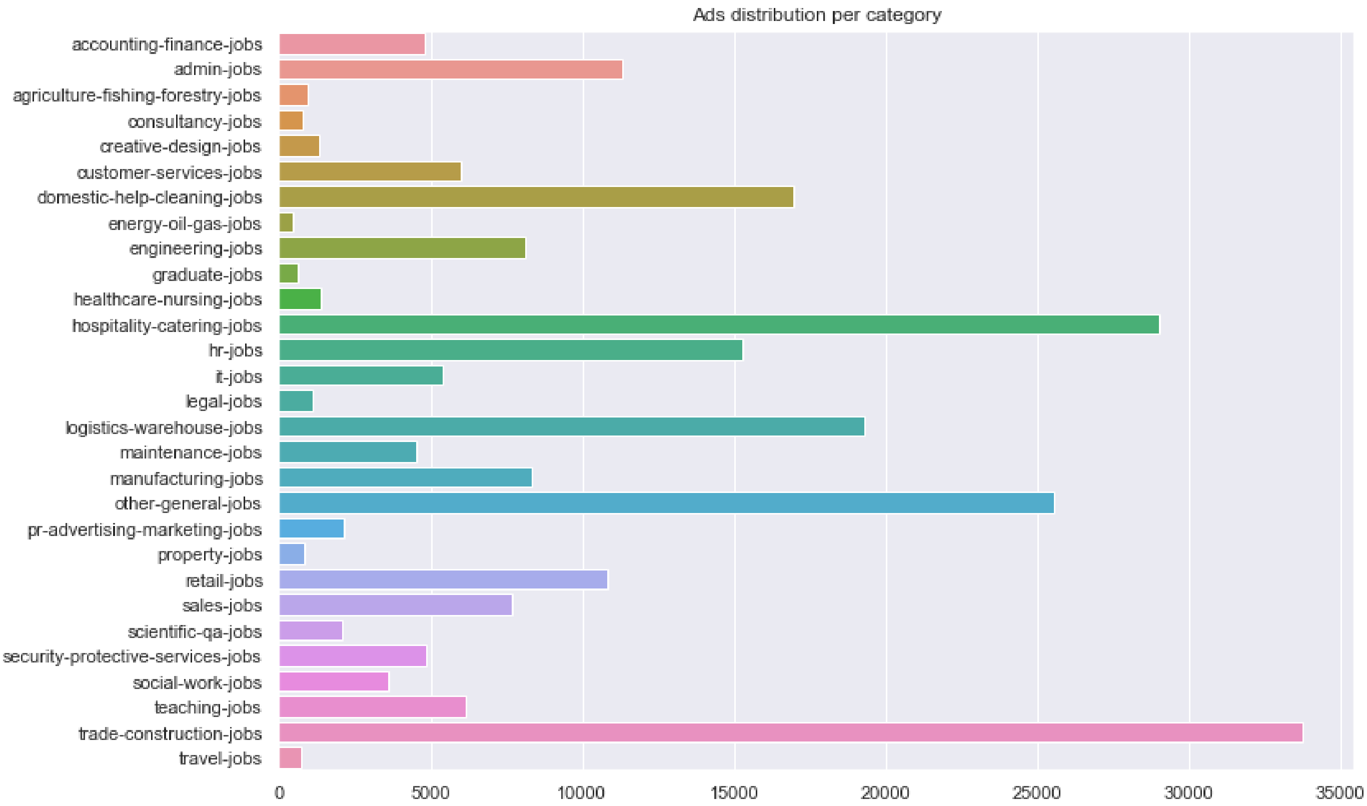


Figure 18: Class distribution of available training and testing Data

As presented in Figure 19 above, there is significant imbalance in the instance amount distribution among the job categories. Specifically, the categories of

Hospitality-Catering, Other-General, Trade-Construction, Logistics-Warehouse, IT, Domestic Help and Admin have 90% of the available Data. On the other hand, there are some categories, such as Consultancy, Travel and Agriculture-Fishing-Forestry which are minority classes with significantly low amount of data. These observations can be useful for the interpretation of later results, in cases of significant differentiations in class wise metrics or overfitting in one or more predominant categories. Finally, the presented distribution among the classes is due to the nature of the UK job market at the time that they were made available for the purposes of this work. Data Augmentation techniques such as under-sampling or synthetic data generation are considered for future work.

Regarding word embeddings, we experimented both with existing pre-trained word vectors as well as custom ones trained on our job advert Data. In the first case, we used the GloVe 100d pre-trained word vectors [49, 62].

5 Methodology

As described previously, 2 different input data representations are examined, namely Bag of Words (BoW) with TF-IDF weights and sequences of words with word embeddings. Depending on the input representation, different Machine Learning and Deep Learning techniques are used. The methodology of processing the Data, transforming them into the desired format and training/tuning the models is described here. In all cases, we split the Data to 72% for training, 20% for testing and 8% for development. In the following cases of non Deep Learning architectures, the Data used were the training set and the test set. In Deep Learning approaches, we also used the development dataset.

5.1 Bag of Words

4 different algorithmic approaches are applied on the Bag of Words representation, namely the Multinomial Naïve Bayes, CART Decision Tree algorithm, Support Vector Classifier and a Feedforward Neural Network. In all following cases, the 4 approaches had the same parameters, which are presented in Table 2.

Approach	Parameter
Multinomial Naïve Bayes	• The additive (Laplace/Lidstone) smoothing parameter alpha was set to 1
Decision Tree	• function to measure the quality of the splits is the Gini impurity • minimum number of samples required to split an internal node was set at 500 • The number of features to consider when looking for the best split was set to the number of input features • No pruning • The strategy used to choose the split at each node was the best split that can be achieved
Support Vector Classifier	• Regularization parameter (C) was set at 0.1 • Linear kernel • Size of kernel cache was set at 200MB • Use of shrinking heuristic
Feedforward	• 2 hidden layers • Activation function in all layers except for output layer is

Neural Network	ReLU • 256 neurons in input layer • 128 neurons in first hidden layer • 64 neurons in second hidden layer • 29 neurons in out put layer (number of categories) • One dropout layer after every hidden layer with dropout value 0.3 • Loss used categorical cross-entropy • Output activation is softmax • Optimizer is Adam

Table 2: Parameters of models used for BoW representations

Each of the described models was used two times: each model was first trained on BoW TF-IDF representations of lowercased job descriptions and secondly on BoW TF-IDF representations of lowercased and stemmed job descriptions. Lowercasing was applied in order to avoid that words with capital letters (starting a sentence etc.) are treated differently when appeared later in a sentence without a capital letter. Stemming is a technique for removing the suffix off a word, thus transforming it to a linguistic root (e.g. the stem of the word “transformer” is “transform”). In both described cases, punctuation and stopwords were removed.

Therefore, the first step here was to preprocess the Data by removing stopwords and punctuation. Then, we continued by firstly just lowercasing the text data and obtaining the first processed dataset and secondly by also stemming the lowercased data and obtaining a second processed dataset. After training the models on the two datasets, we obtained 8 trained models in total.

5.2 Sequences of words and pre-trained embeddings

After finishing with BoW representation, we experimented with word embeddings and padded sequences. The preprocessing steps here were lowercasing and stemming (with removal of stopwords and punctuation). First, we used pre-trained word embeddings, namely Google’s word2vec embeddings (Glove 100d). After transforming the text data into the described format, we trained three different deep learning architectures on them, namely a Feedforward Neural Network, a Convolutional Neural Network and a Recurrent Neural Network. The parameters of these models are presented in Table 3. After this process, we obtained another three trained models.

Approach	Parameter
Feedforward Neural Network	• 3 hidden layers • The length of the embedding vector is 100 • Activation function in all layers except for output layer is ReLU • Embedding layer serves as input layer • 512 neurons in first hidden layer • 256 neurons in second hidden layer • 128 neurons in third hidden layer • 29 neurons in output layer (number of categories) • One dropout layer after every hidden layer with dropout value 0.5 • Loss used categorical cross-entropy • Output activation is softmax • Optimizer is Adam
Convolutional Neural Network	• Embedding layer serves as input layer (length of embedding vector is 100) • 3 parallel convolutional layers with 1, 5, 10 window sizes (filters) respectively • After the first two convolutional layers, there is a max pooling function with pooling size 5 • After the 3rd convolutional layer, there is a max pooling function with pooling size 30 • There are two more convolutional layers with window sizes (filters) 3 • All convolutional layers are of size 128 • dense layer (128 neurons) • output dense layer (20 neurons) • Activation function in all layers except for output layer is ReLU • Loss used categorical cross-entropy • Output activation is softmax • Optimizer is Adam
Recurrent Neural Network	• Embedding layer serves as input layer (length of embedding vector is 100) • LSTM layer with 64 LSTM units • Dense layer with 32 neurons with ReLU activation • Output dense layer with 29 neurons • Loss used categorical cross-entropy • Output activation is softmax • Optimizer is Adam

Table 3: Parameters of trained models on sequences of words

5.3 Sequences of words and custom embeddings

Finally, we trained our own custom embeddings and re-trained the three deep learning architectures of Table 3. Specifically, we used the skipgram algorithm to train custom word2vec embeddings on the lowercased dataset of the first step. In addition, we used the skipgram algorithm to train custom FastText embeddings on the lowercased dataset of the first step. After obtaining the two custom embeddings (word2vec and FastText), we applied the three Deep Learning models of Table 2 on them, which resulted in another 6 models. The Table 4 summarizes all trained models.

Model	Input Data Representation
MNB	BoW (TF-IDF) on lowercased job descriptions
MNB	BoW (TF-IDF) on lowercased and stemmed job descriptions
DT	BoW (TF-IDF) on lowercased job descriptions
DT	BoW (TF-IDF) on lowercased and stemmed job descriptions
SVC	BoW (TF-IDF) on lowercased job descriptions
SVC	BoW (TF-IDF) on lowercased and stemmed job descriptions
FNN	BoW (TF-IDF) on lowercased job descriptions
FNN	BoW (TF-IDF) on lowercased and stemmed job descriptions
FNN	Lowercased job descriptions and Glove word embedding layer
FNN	Lowercased job descriptions and custom word2vec embedding layer
FNN	Lowercased job descriptions and custom FastText embedding layer
CNN	Lowercased job descriptions and Glove word embedding layer
CNN	Lowercased job descriptions and custom word2vec embedding layer
CNN	Lowercased job descriptions and custom FastText embedding layer
RNN	Lowercased job descriptions and Glove word embedding layer
RNN	Lowercased job descriptions and custom word2vec embedding layer
RNN	Lowercased job descriptions and custom FastText embedding layer

Table 4: Summary of trained models

5.4 Evaluation metrics

All models were evaluated with use of Recall, Precision, F1-Score and Accuracy both class-wise and with averaging over all classes. If we define as True Positives (TP) of a class X the number of correct classifications (predicted as X while being actually X), as False Positives (FP) of a class X the amount of predictions X that were not actually X, as False Negatives (FN) of a class X the number of predictions not equal to X that were actually X and as True Negatives (TN) of a class X the number of predictions not equal to X that were actually not X, then Precision, Recall and F1-Score of the Class X are defined as:

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN} \qquad F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Precision is used to measure how accurate the model is in predicting positives, which means how many positive predictions of the Class X were indeed positive. High Precision is needed when the cost of False Positives is high, i.e. when there is a high need not predict Negatives as Positives (e.g. spam detection). Recall measures how many of the actual Positives were classified as Positives. High Recall is needed when there is a high need of not having False Negatives (e.g. sick patient detection). F1-Score is a measure that is used when there is a need to balance between good Precision and good Recall. For the purposes of evaluating the models, we used Precision, Recall and F1-Score of each class as well as the testing and training accuracies of the model. Accuracy is defined as:

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

For the purposed of the models' evaluation, we also used the macro average and weighted average of the Recall, Precision and F1-Scores of all classes within a model. Macro average for a metric measures the average value of the metric for all classes. The weighted average calculates the average metric's value using a weight that depends on the number of true labels for each class: For instance, Macro F1 and weighted F1 are:

$$macro\ average\ F1 = \frac{\sum_{i=1}^{n} F1_i}{n} \qquad weighted\ average\ F1 = \frac{\sum_{i=1}^{n} F1_i W_i}{n}$$

6 Results and Discussion

6.1 Classification results per model (Accuracy, Recall, Precision and F1-Score)

1) Multinomial Naïve Bayes trained on Bag of Words TF-IDF with lower case descriptions

The Multinomial Naïve Bayes on BoW representation of lower-case descriptions achieved a training accuracy of 73.7% and a testing accuracy of 72.1%. The macro average Precision, Recall and F1-Score were 65%, 48% and 49% respectively. The weighted average Precision, Recall and F1-Score were 72%, 72% and 72% respectively. The Precision, Recall and F1-Score of each class are highlighted in Figure 19. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 5 below:

	Precision	**Recall**	**F1-Score**
Macro average	65%	48%	49%
Weighted average	72%	72%	72%

Table 5: Average metrics of MNB on BoW and lower case descriptions

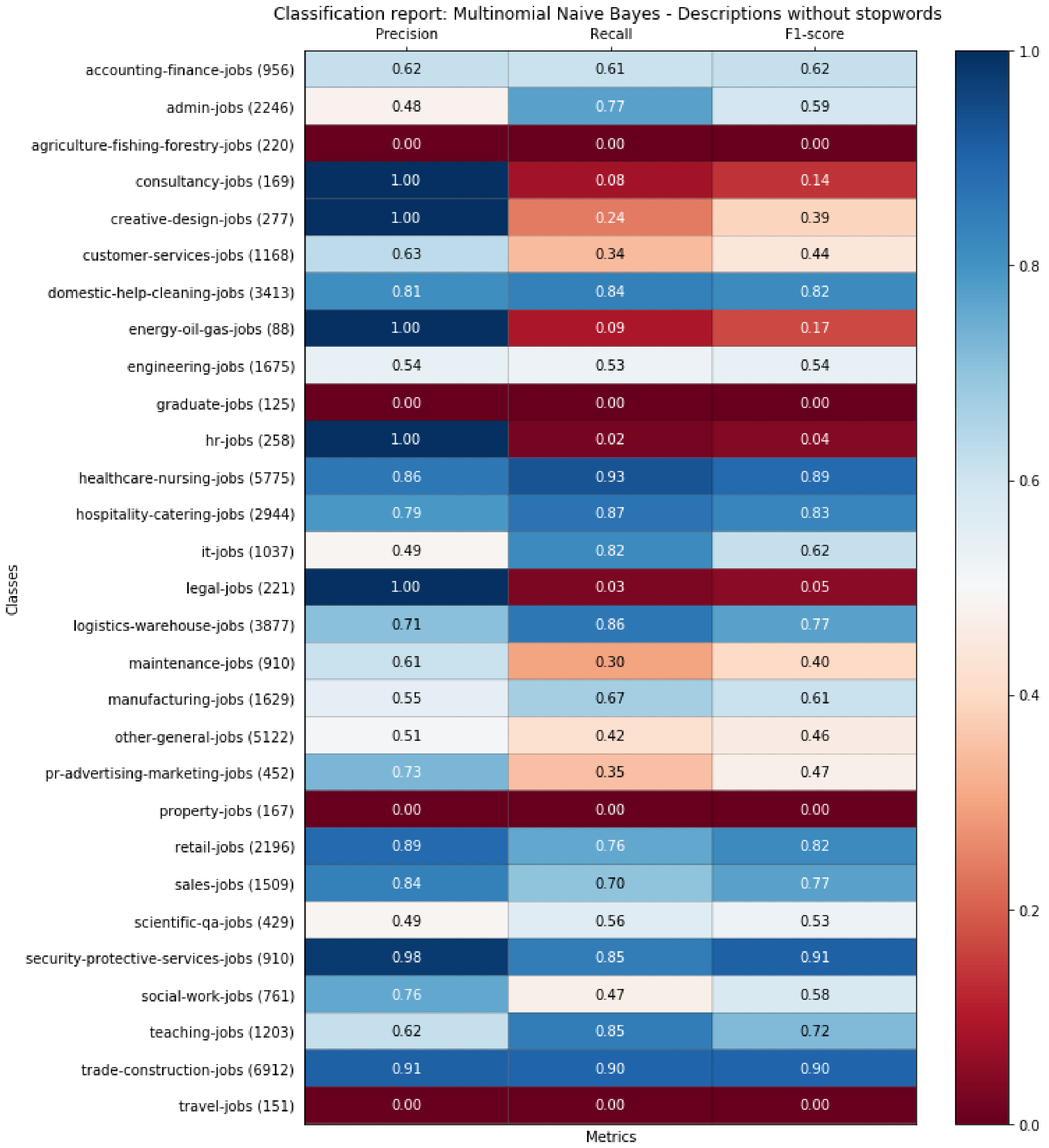


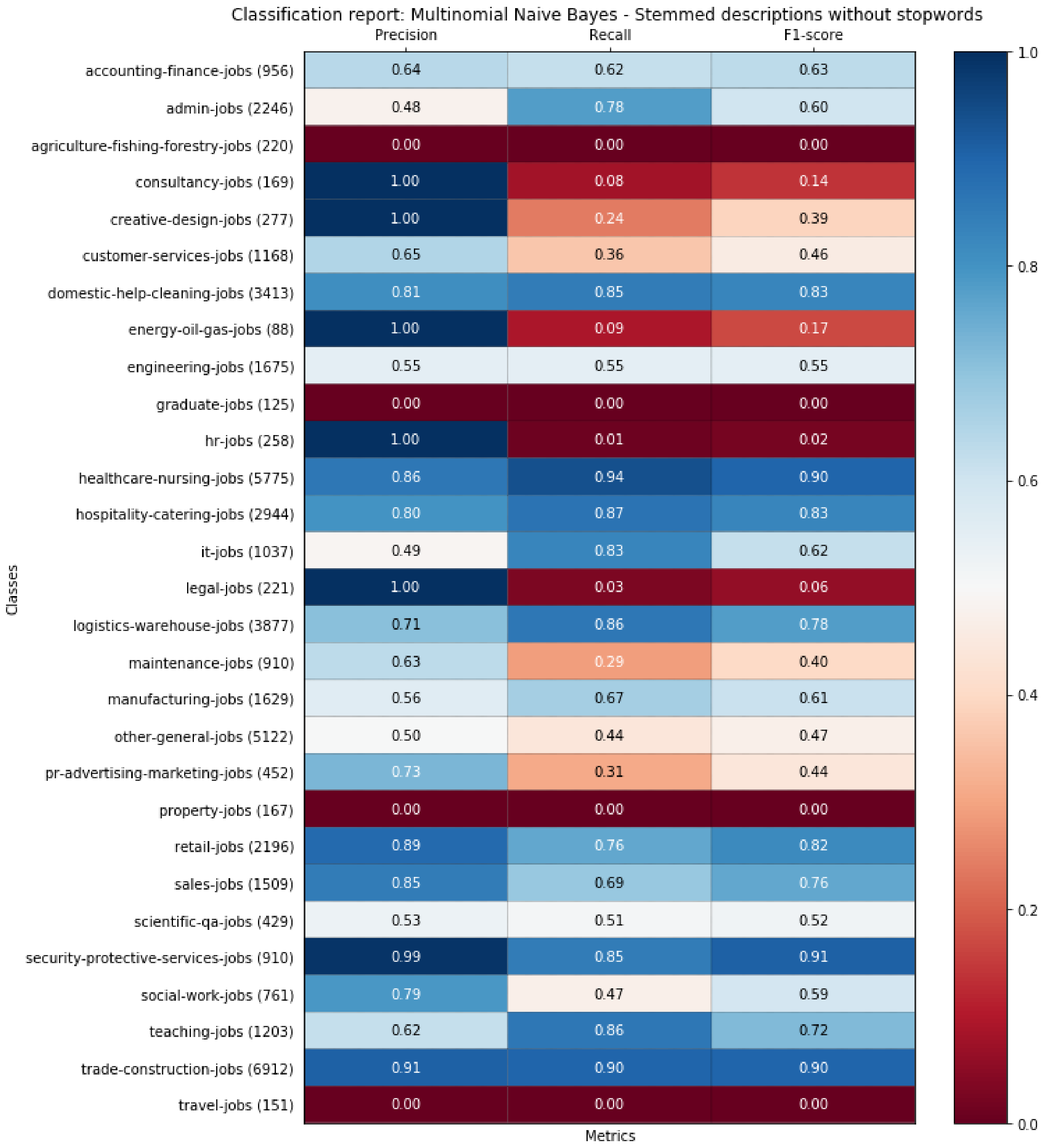
Figure 19: Classification Report of MNB on BoW and lower case descriptions

2) Multinomial Naïve Bayes trained on Bag of Words TF-IDF with lower case stemmed descriptions

The Multinomial Naïve Bayes on BoW representation of lower case and stemmed descriptions achieved a training accuracy of 73.8% and a testing accuracy of 72.4%. The macro average Precision, Recall and F1-Score were 65%, 48% and 49% respectively. The weighted average Precision, Recall and F1-Score were 73%, 72% and 71% respectively. The Precision, Recall and F1-Score of each class are highlighted in Figure 20. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 6 below:

	Precision	**Recall**	**F1-Score**
Macro average	65%	48%	49%
Weighted average	73%	72%	71%

Table 6: Average metrics of MNB on BoW and lower case and stemmed descriptions



Classes	Precision	Recall	F1-score
accounting-finance-jobs (956)	0.64	0.62	0.63
admin-jobs (2246)	0.48	0.78	0.60
agriculture-fishing-forestry-jobs (220)	0.00	0.00	0.00
consultancy-jobs (169)	1.00	0.08	0.14
creative-design-jobs (277)	1.00	0.24	0.39
customer-services-jobs (1168)	0.65	0.36	0.46
domestic-help-cleaning-jobs (3413)	0.81	0.85	0.83
energy-oil-gas-jobs (88)	1.00	0.09	0.17
engineering-jobs (1675)	0.55	0.55	0.55
graduate-jobs (125)	0.00	0.00	0.00
hr-jobs (258)	1.00	0.01	0.02
healthcare-nursing-jobs (5775)	0.86	0.94	0.90
hospitality-catering-jobs (2944)	0.80	0.87	0.83
it-jobs (1037)	0.49	0.83	0.62
legal-jobs (221)	1.00	0.03	0.06
logistics-warehouse-jobs (3877)	0.71	0.86	0.78
maintenance-jobs (910)	0.63	0.29	0.40
manufacturing-jobs (1629)	0.56	0.67	0.61
other-general-jobs (5122)	0.50	0.44	0.47
pr-advertising-marketing-jobs (452)	0.73	0.31	0.44
property-jobs (167)	0.00	0.00	0.00
retail-jobs (2196)	0.89	0.76	0.82
sales-jobs (1509)	0.85	0.69	0.76
scientific-qa-jobs (429)	0.53	0.51	0.52
security-protective-services-jobs (910)	0.99	0.85	0.91
social-work-jobs (761)	0.79	0.47	0.59
teaching-jobs (1203)	0.62	0.86	0.72
trade-construction-jobs (6912)	0.91	0.90	0.90
travel-jobs (151)	0.00	0.00	0.00

Figure 20: Classification Report of MNB on BoW and lower case and stemmed descriptions

3) Decision Tree trained on Bag of Words TF-IDF with lower case descriptions

The Decision Tree algorithm on BoW representation of lower case descriptions achieved a training accuracy of 74.1% and a testing accuracy of 68.1%. The macro average Precision, Recall and F1-Score were 57%, 48% and 50% respectively. The weighted average Precision, Recall and F1-Score were 68%, 68% and 68% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 21. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 7 below:

	Precision	**Recall**	**F1-Score**
Macro average	57%	48%	50%
Weighted average	68%	68%	68%

Table 7: Average metrics of Decision Tree on BoW and lower case and stemmed descriptions

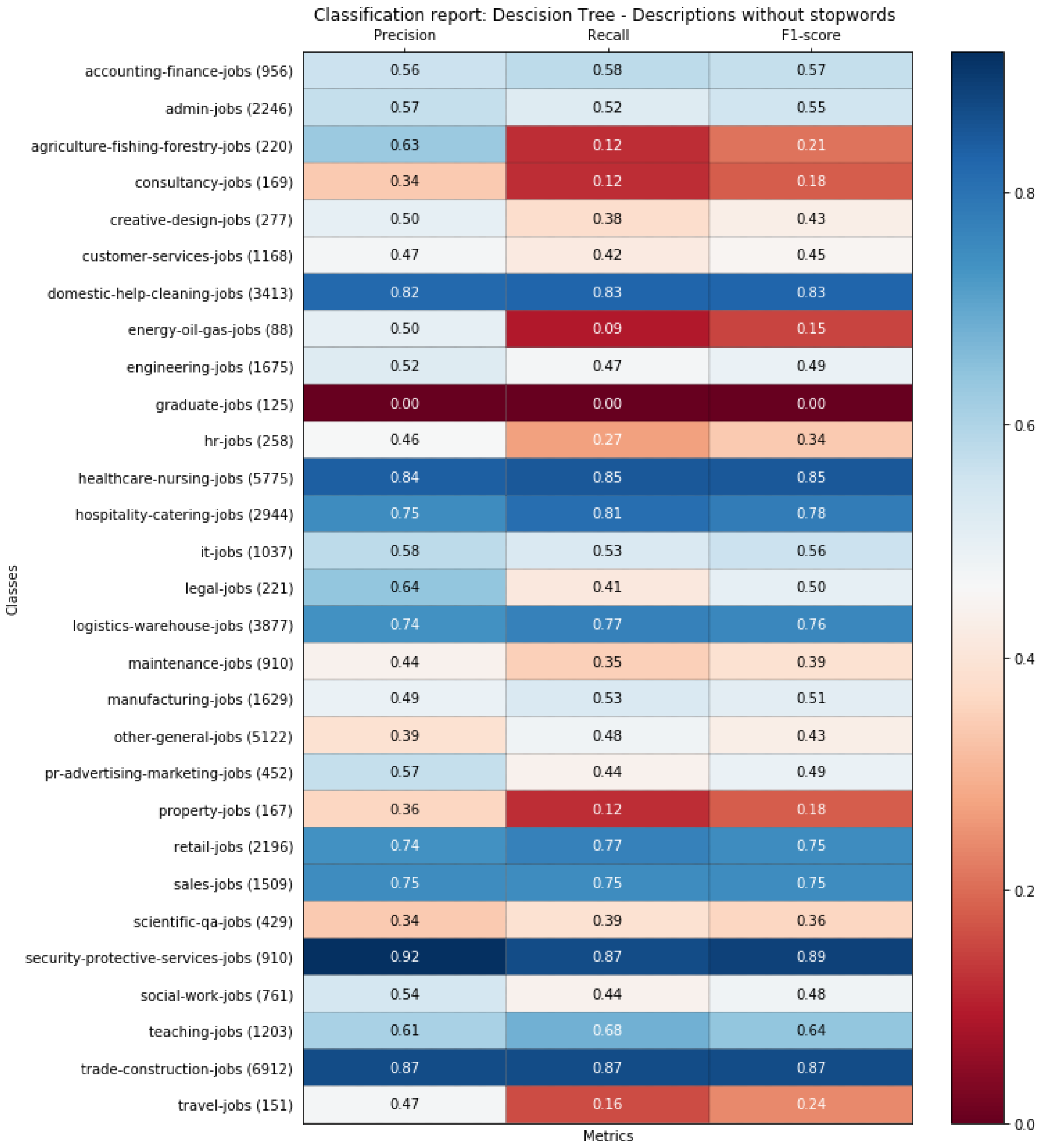


Figure 21: Classification Report of Decision Tree on BoW and lower case descriptions

4) Decision Tree trained on Bag of Words TF-IDF with lower case and stemmed descriptions

The Decision Tree algorithm on BoW representation of lower case and stemmed descriptions achieved a training accuracy of 73.1% and a testing accuracy of 67.6%. The macro average Precision, Recall and F1-Score were 59%, 48% and 50% respectively. The weighted average Precision, Recall and F1-Score were 68%, 68% and 67% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 22. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 8 below:

	Precision	**Recall**	**F1-Score**
Macro average	59%	48%	50%
Weighted average	68%	68%	67%

Table 8: Average metrics of Decision Tree on BoW and lower case and stemmed descriptions

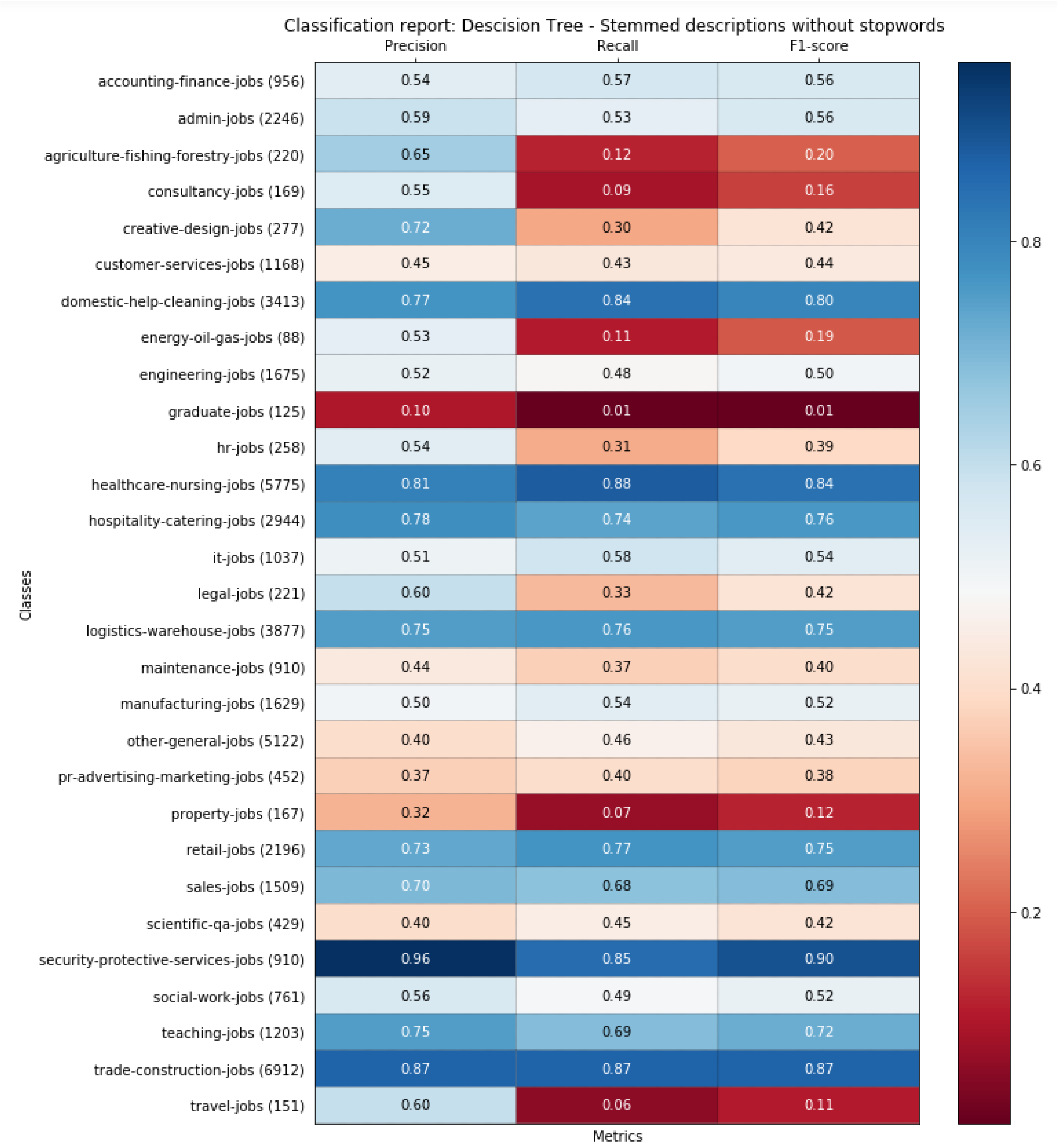


Figure 22: Classification Report of Decision Tree on BoW and lower case and stemmed descriptions

5) Support Vector Classifier with linear kernel trained on Bag of Words TF-IDF with lower case descriptions

The Support Vector Classifer on BoW representation of lower case descriptions achieved a training accuracy of 79.5% and a testing accuracy of 77.3%. The macro average Precision, Recall and F1-Score were 73%, 56% and 59% respectively. The weighted average Precision, Recall and F1-Score were 78%, 77% and 77% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 23. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 9 below:

	Precision	Recall	F1-Score
Macro average	73%	56%	59%
Weighted average	78%	77%	77%

Table 9: Average metrics of SVC on BoW and lower case descriptions

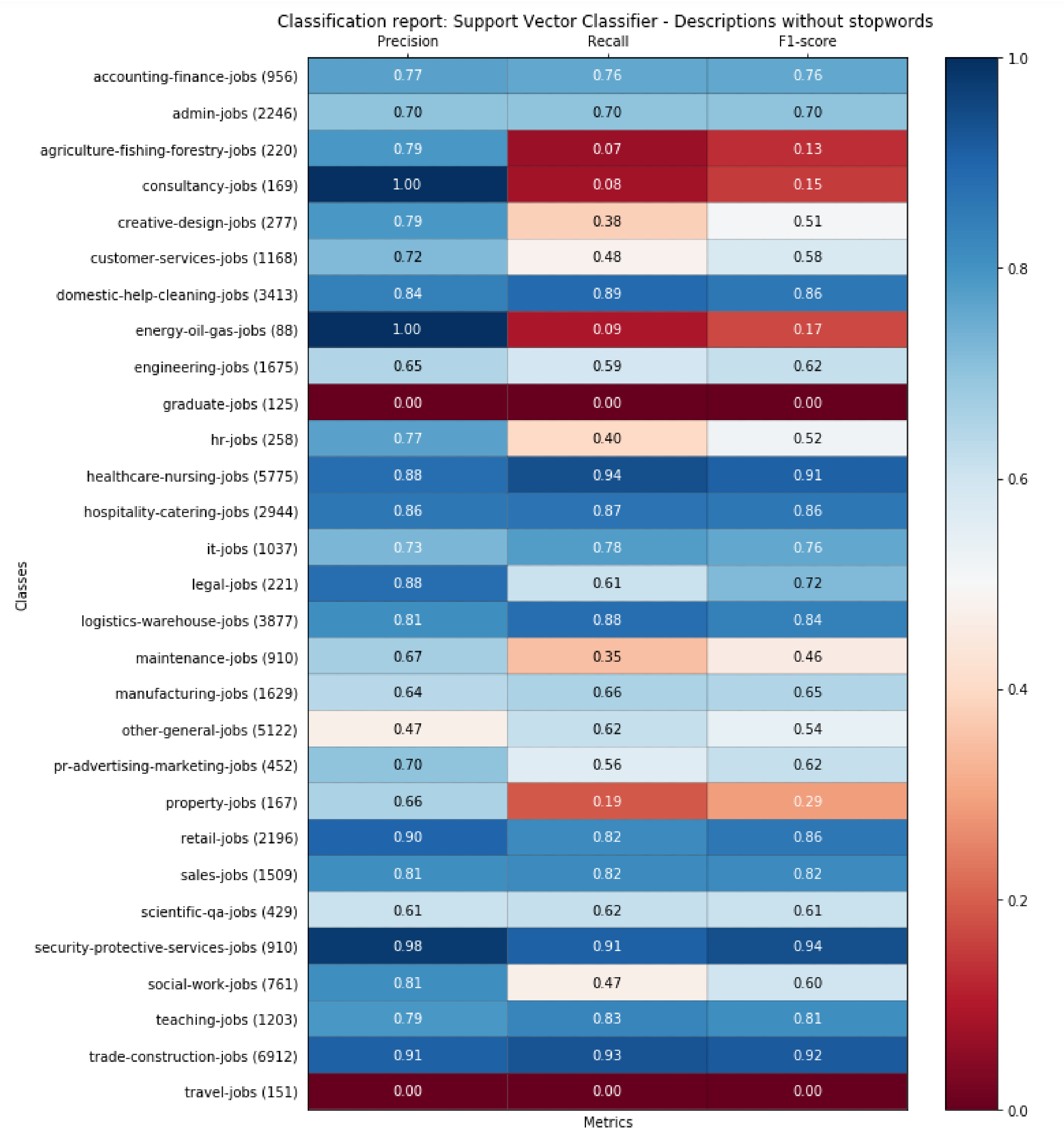


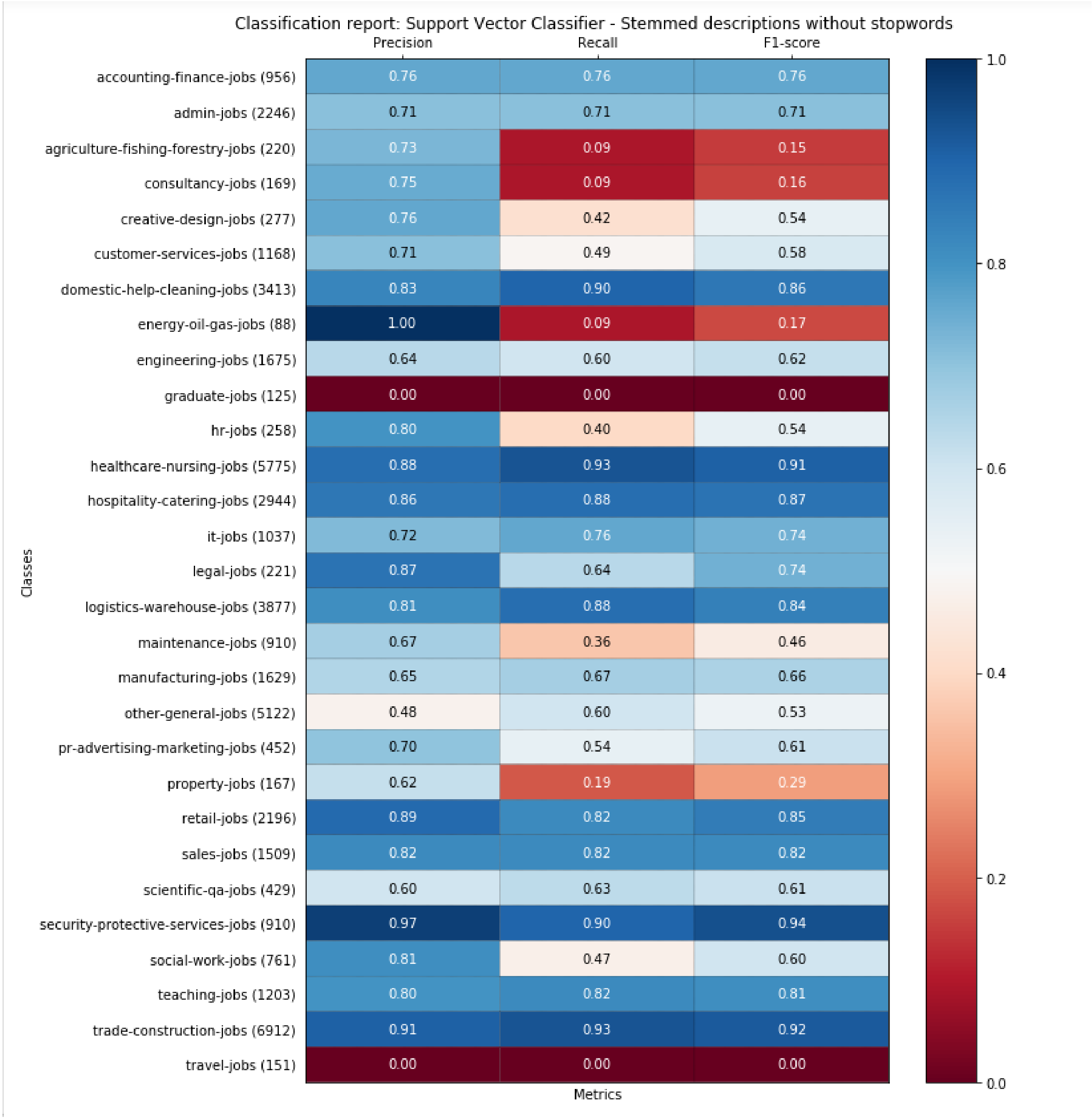
Figure 23: Classification Report of SVC on BoW and lower case descriptions

6) Support Vector Classifier with linear kernel trained on Bag of Words TF-IDF with lower case and stemmed descriptions

The Support Vector Classifer on BoW representation of lower case descriptions achieved a training accuracy of 79.5% and a testing accuracy of 77.3%. The macro average Precision, Recall and F1-Score were 72%, 57% and 60% respectively. The weighted average Precision, Recall and F1-Score were 78%, 77% and 77% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 24. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 10 below:

	Precision	**Recall**	**F1-Score**
Macro average	72%	57%	60%
Weighted average	78%	77%	77%

Table 10: Average Metrics of SVC on BoW and lower case and stemmed descriptions



Classification report: Support Vector Classifier - Stemmed descriptions without stopwords

Classes	Precision	Recall	F1-score
accounting-finance-jobs (956)	0.76	0.76	0.76
admin-jobs (2246)	0.71	0.71	0.71
agriculture-fishing-forestry-jobs (220)	0.73	0.09	0.15
consultancy-jobs (169)	0.75	0.09	0.16
creative-design-jobs (277)	0.76	0.42	0.54
customer-services-jobs (1168)	0.71	0.49	0.58
domestic-help-cleaning-jobs (3413)	0.83	0.90	0.86
energy-oil-gas-jobs (88)	1.00	0.09	0.17
engineering-jobs (1675)	0.64	0.60	0.62
graduate-jobs (125)	0.00	0.00	0.00
hr-jobs (258)	0.80	0.40	0.54
healthcare-nursing-jobs (5775)	0.88	0.93	0.91
hospitality-catering-jobs (2944)	0.86	0.88	0.87
it-jobs (1037)	0.72	0.76	0.74
legal-jobs (221)	0.87	0.64	0.74
logistics-warehouse-jobs (3877)	0.81	0.88	0.84
maintenance-jobs (910)	0.67	0.36	0.46
manufacturing-jobs (1629)	0.65	0.67	0.66
other-general-jobs (5122)	0.48	0.60	0.53
pr-advertising-marketing-jobs (452)	0.70	0.54	0.61
property-jobs (167)	0.62	0.19	0.29
retail-jobs (2196)	0.89	0.82	0.85
sales-jobs (1509)	0.82	0.82	0.82
scientific-qa-jobs (429)	0.60	0.63	0.61
security-protective-services-jobs (910)	0.97	0.90	0.94
social-work-jobs (761)	0.81	0.47	0.60
teaching-jobs (1203)	0.80	0.82	0.81
trade-construction-jobs (6912)	0.91	0.93	0.92
travel-jobs (151)	0.00	0.00	0.00

Figure 24: Classification Report of SVC on BoW and lower case and stemmed descriptions

7) Feedforward Neural Network trained on Bag of Words TF-IDF with lower case descriptions

The Feedforward Neural Network on BoW representation of lower case descriptions achieved a training accuracy of 79.5% and a testing accuracy of 80.37%. The macro average Precision, Recall and F1-Score were 74%, 64% and 65% respectively. The weighted average Precision, Recall and F1-Score were 80%, 80% and 80% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 27. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 11 below:

	Precision	**Recall**	**F1-Score**
Macro average	74%	64%	65%
Weighted average	80%	80%	80%

Table 11: Average Metrics of FNN on Bow and lower case descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 26 and 27 respectively.

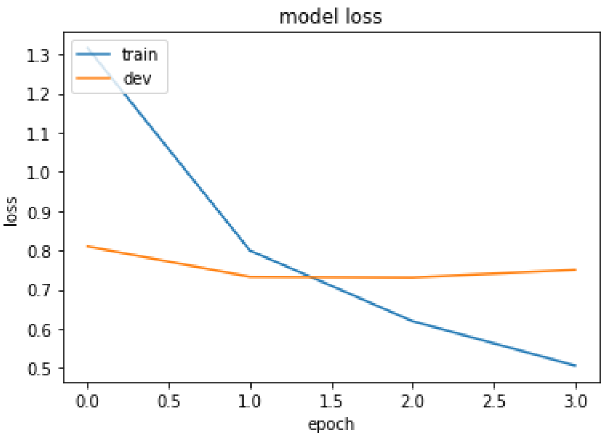


Figure 25: Loss per training epoch for FNN with lower case descriptions

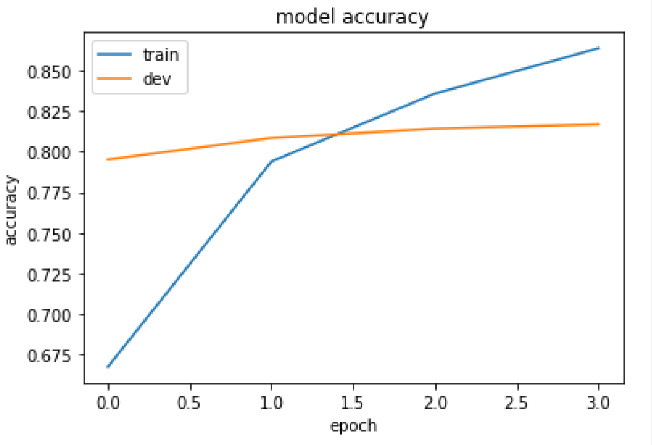


Figure 26: Accuracy per training epoch for FNN with lower case descriptions

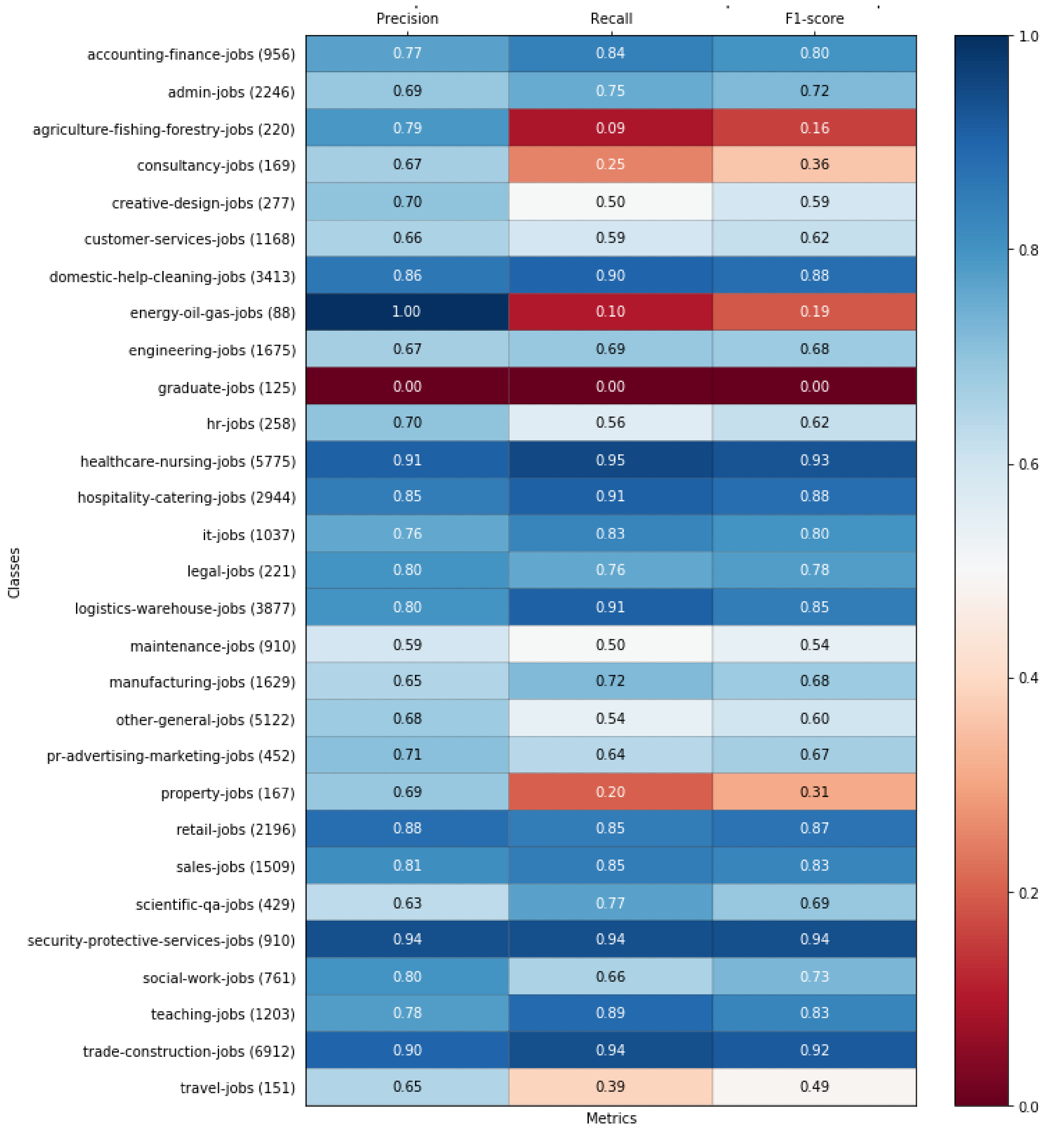


Figure 27: Classification Report for FNN with lower case descriptions

8) Feedforward Neural Network trained on Bag of Words TF-IDF with lower case and stemmed descriptions

The Feedforward Neural Network on BoW representation of lower case and stemmed descriptions achieved a training accuracy of 79.5% and a testing accuracy of 80.5%. The macro average Precision, Recall and F1-Score were 76%, 63% and 65% respectively. The weighted average Precision, Recall and F1-Score were 80%, 81% and 80% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 30. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 12 below:

	Precision	**Recall**	**F1-Score**
Macro average	76%	63%	65%
Weighted average	80%	81%	80%

Table 12: Average Metrics of FNN on BoW and lowercased and stemmed descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 28 and 29 respectively.

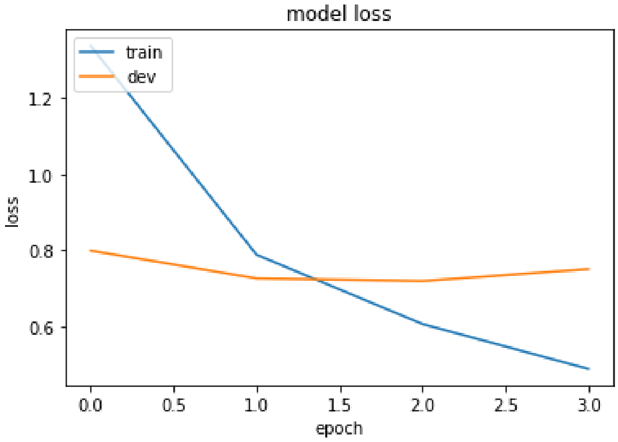


Figure 28: Loss per training epoch for FNN with lower case and stemmed descriptions

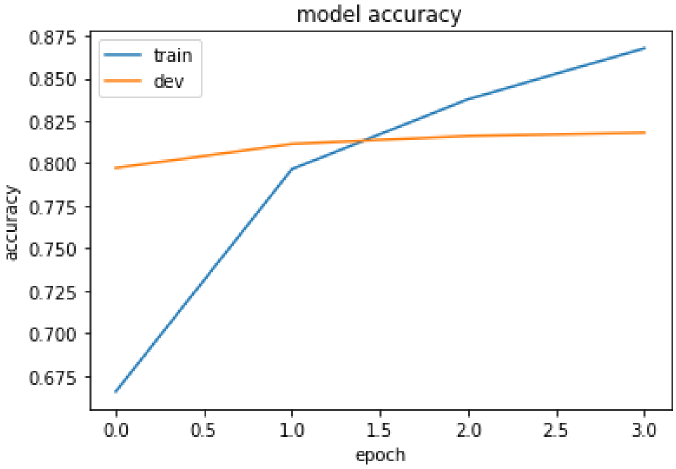
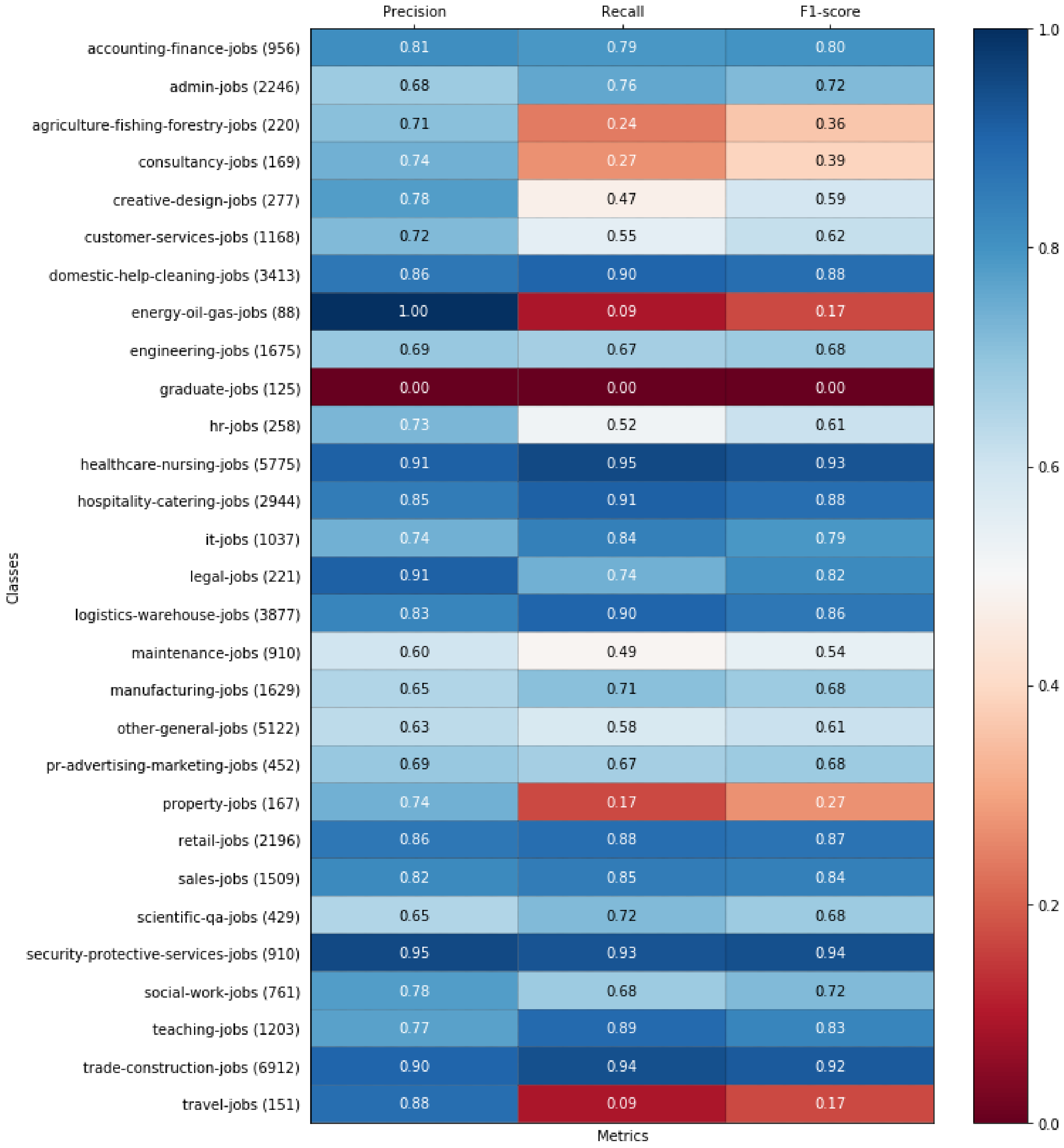


Figure 29: Accuracy per training epoch for FNN with lower case and stemmed descriptions



Classes	Precision	Recall	F1-score
accounting-finance-jobs (956)	0.81	0.79	0.80
admin-jobs (2246)	0.68	0.76	0.72
agriculture-fishing-forestry-jobs (220)	0.71	0.24	0.36
consultancy-jobs (169)	0.74	0.27	0.39
creative-design-jobs (277)	0.78	0.47	0.59
customer-services-jobs (1168)	0.72	0.55	0.62
domestic-help-cleaning-jobs (3413)	0.86	0.90	0.88
energy-oil-gas-jobs (88)	1.00	0.09	0.17
engineering-jobs (1675)	0.69	0.67	0.68
graduate-jobs (125)	0.00	0.00	0.00
hr-jobs (258)	0.73	0.52	0.61
healthcare-nursing-jobs (5775)	0.91	0.95	0.93
hospitality-catering-jobs (2944)	0.85	0.91	0.88
it-jobs (1037)	0.74	0.84	0.79
legal-jobs (221)	0.91	0.74	0.82
logistics-warehouse-jobs (3877)	0.83	0.90	0.86
maintenance-jobs (910)	0.60	0.49	0.54
manufacturing-jobs (1629)	0.65	0.71	0.68
other-general-jobs (5122)	0.63	0.58	0.61
pr-advertising-marketing-jobs (452)	0.69	0.67	0.68
property-jobs (167)	0.74	0.17	0.27
retail-jobs (2196)	0.86	0.88	0.87
sales-jobs (1509)	0.82	0.85	0.84
scientific-qa-jobs (429)	0.65	0.72	0.68
security-protective-services-jobs (910)	0.95	0.93	0.94
social-work-jobs (761)	0.78	0.68	0.72
teaching-jobs (1203)	0.77	0.89	0.83
trade-construction-jobs (6912)	0.90	0.94	0.92
travel-jobs (151)	0.88	0.09	0.17

Figure 30: Classification Report for FNN with lower case and stemmed descriptions

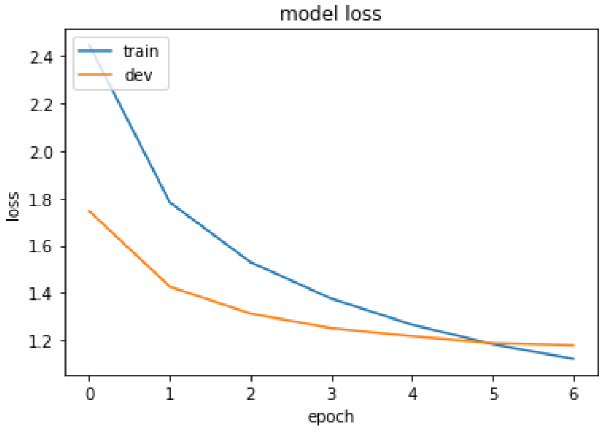
9) Feedforward Neural Network trained on Google's word2vec (Glove) embeddings with lower case descriptions

The Feedforward Neural Network on sequences of words and on Google's word2vec (Glove) embeddings of lower case descriptions achieved a training accuracy of 66.86% and a testing accuracy of 66.63%. The macro average Precision, Recall and F1-Score were 42%, 40% and 40% respectively. The weighted average Precision, Recall and F1-Score were 64%, 67% and 64% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 33. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 13 below:

	Precision	**Recall**	**F1-Score**
Macro average	42%	40%	40%
Weighted average	64%	67%	64%

Table 13: Average Metrics for FNN on Google's word2vec (Glove) embeddings with lower case descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 31 and 32 respectively.



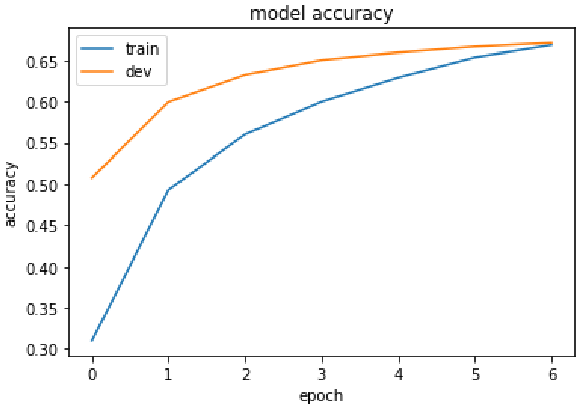


Figure 31: Loss per training epoch for FNN on Google's word2vec (Glove) embeddings with lower case descriptions

Figure 32: Accuracy per training epoch for FNN on Google's word2vec (Glove) embeddings with lower case descriptions

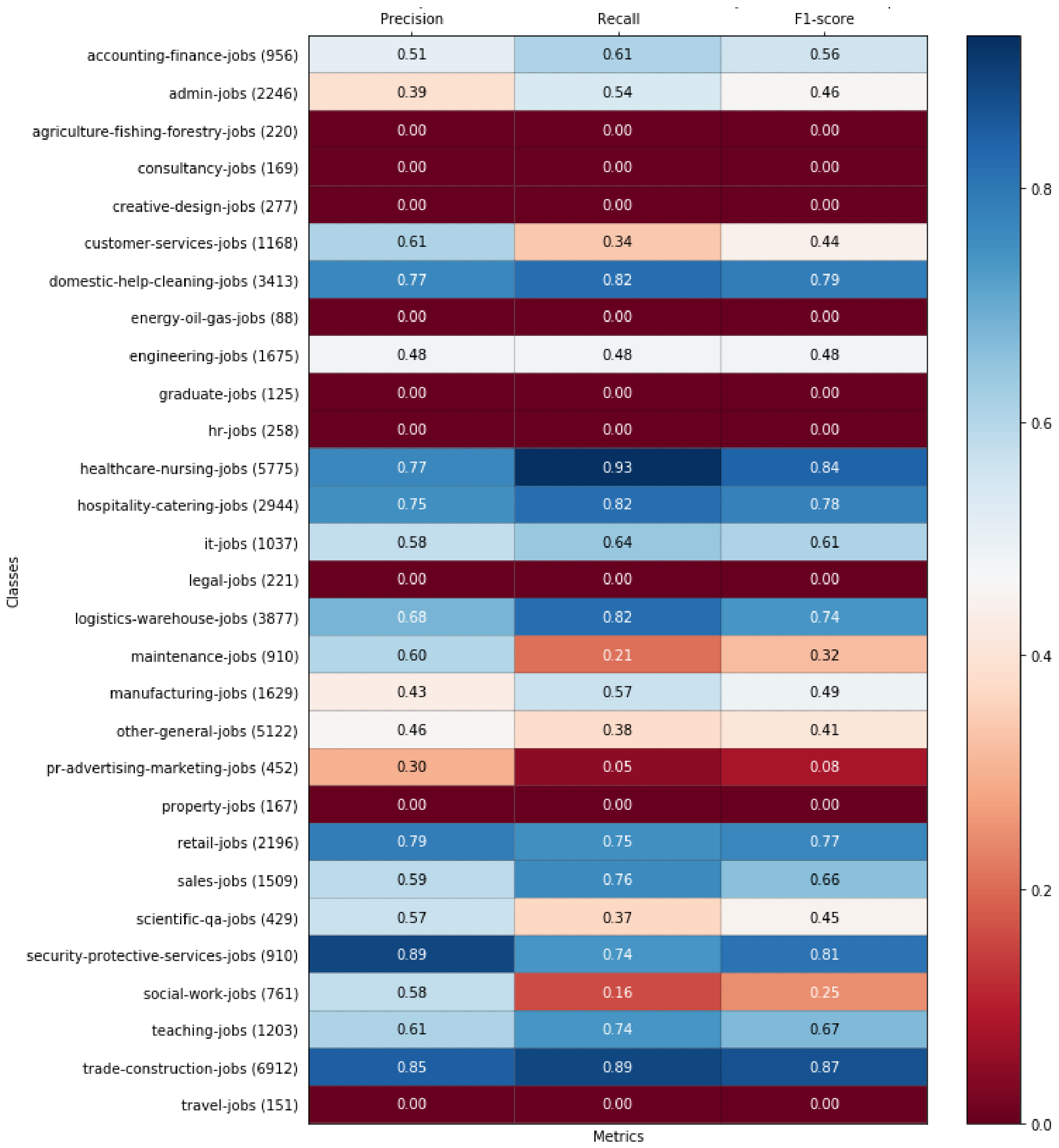


Figure 33: Classification Report for FNN on Google's word2vec (Glove) embeddings with lower case descriptions

10) Feedforward Neural Network trained on custom word2vec embeddings (with skipgram algorithm) with lower case descriptions

The Feedforward Neural Network on sequences of words and on custom word2vec embeddings of lower case descriptions achieved a training accuracy of 73.25% and a testing accuracy of 73.66%. The macro average Precision, Recall and F1-Score were 52%, 48% and 48% respectively. The weighted average Precision, Recall and F1-Score were 72%, 74% and 72% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 36. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 14 below:

	Precision	**Recall**	**F1-Score**
Macro average	52%	48%	48%
Weighted average	72%	74%	72%

Table 14: Average Metrics for FNN custom word2vec embeddings (skipgram) with lower case descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 34 and 35 respectively.

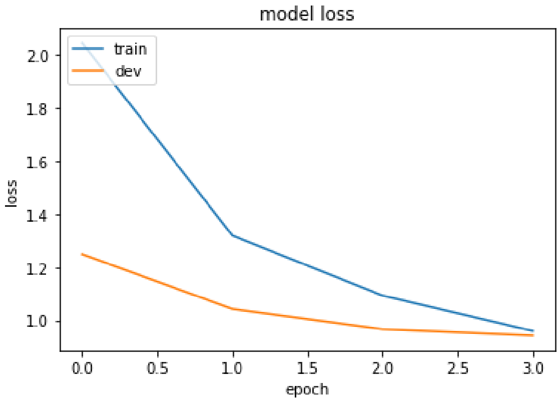


Figure 34: Loss per training epoch for FNN custom word2vec embeddings (skipgram) with lower case descriptions

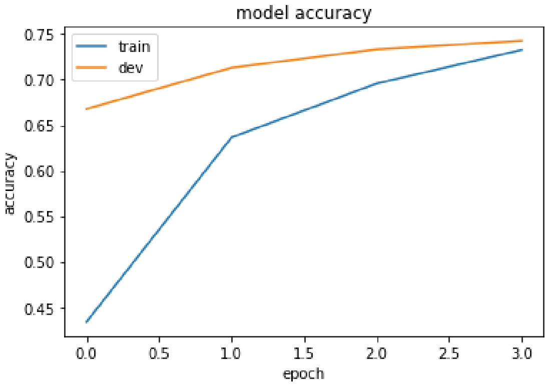


Figure 35: Accuracy per training epoch for FNN custom word2vec embeddings (skipgram) with lower case descriptions

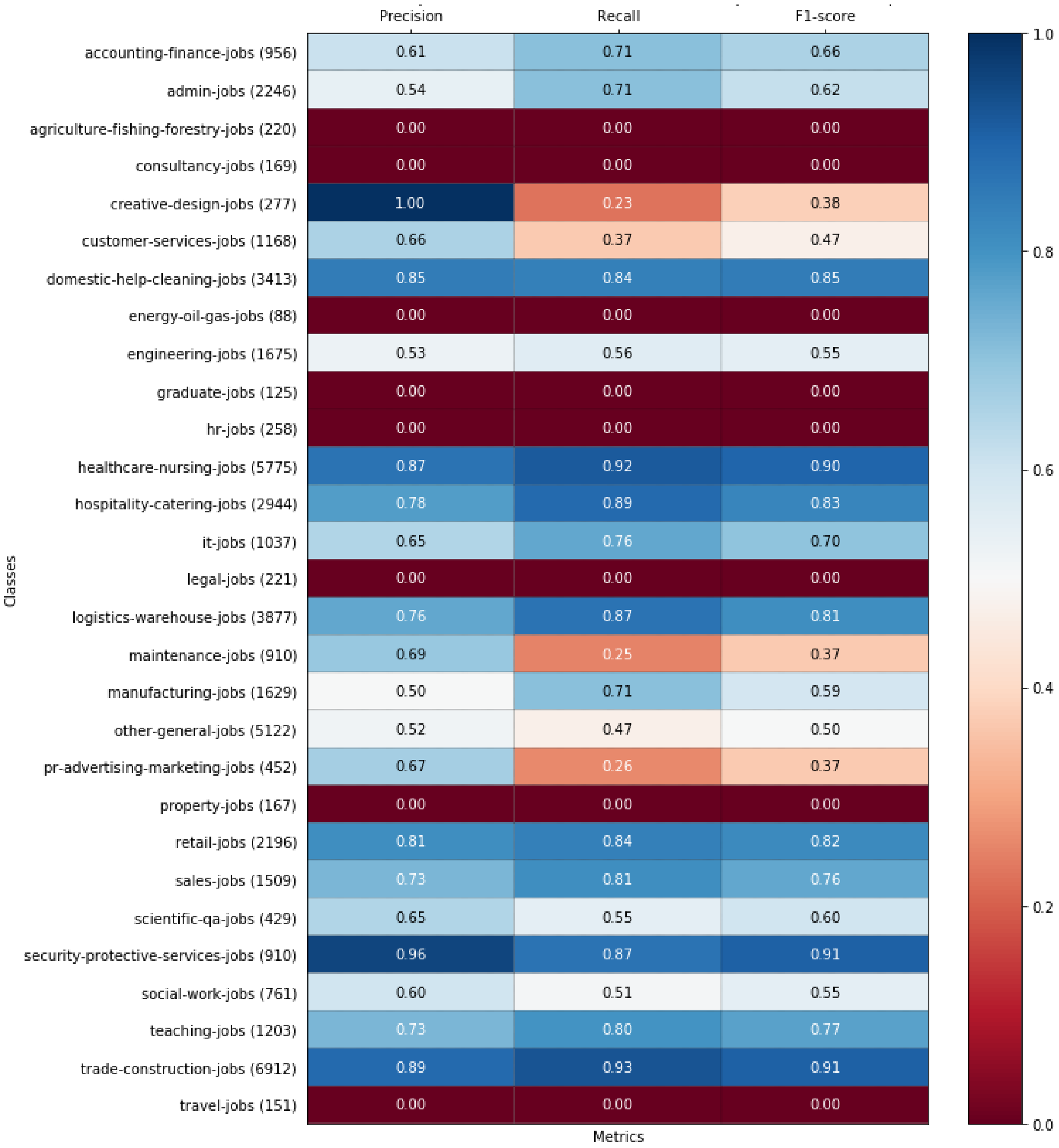


Figure 36: Classification Report for FNN custom word2vec embeddings (skipgram) with lower case descriptions

11) Feedforward Neural Network trained on custom FastText embeddings (with skipgram algorithm) with lower case descriptions

The Feedforward Neural Network on sequences of words and on custom FastText embeddings of lower case descriptions achieved a training accuracy of 73.38% and a testing accuracy of 72.65%. The macro average Precision, Recall and F1-Score were 47%, 47% and 47% respectively. The weighted average Precision, Recall and F1-Score were 71%, 73% and 72% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 39. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 15 below:

	Precision	**Recall**	**F1-Score**
Macro average	47%	47%	47%
Weighted average	71%	73%	72%

Table 15: Average Metrics for FNN custom word2vec embeddings (skipgram) with lower case descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 37 and 38 respectively.

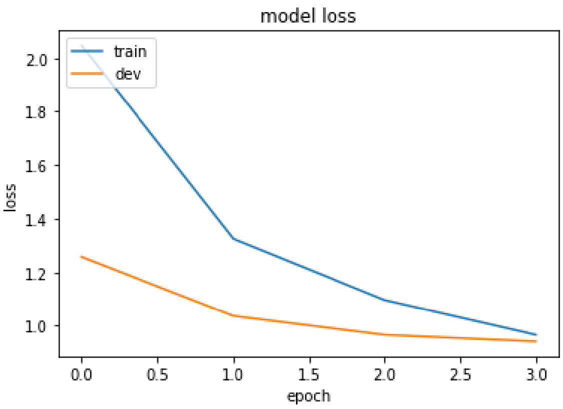


Figure 37: Loss per training epoch for FNN with custom FastText embeddings (skipgram) with lower case descriptions

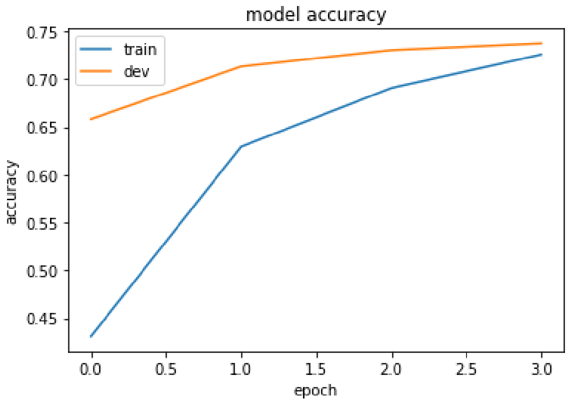
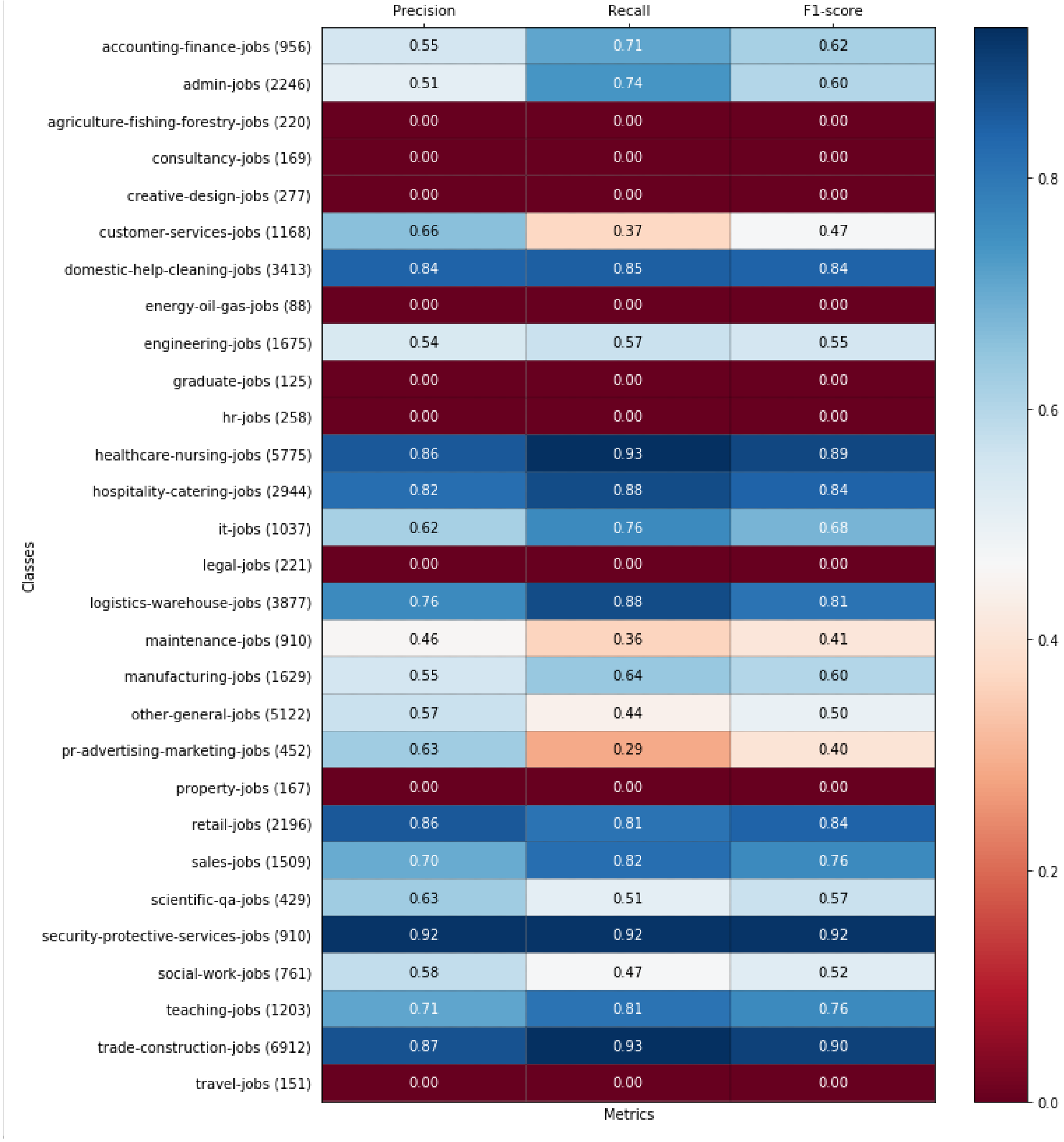


Figure 38: Accuracy per training epoch for FNN with custom FastText embeddings (skipgram) with lower case descriptions



Classes	Precision	Recall	F1-score
accounting-finance-jobs (956)	0.55	0.71	0.62
admin-jobs (2246)	0.51	0.74	0.60
agriculture-fishing-forestry-jobs (220)	0.00	0.00	0.00
consultancy-jobs (169)	0.00	0.00	0.00
creative-design-jobs (277)	0.00	0.00	0.00
customer-services-jobs (1168)	0.66	0.37	0.47
domestic-help-cleaning-jobs (3413)	0.84	0.85	0.84
energy-oil-gas-jobs (88)	0.00	0.00	0.00
engineering-jobs (1675)	0.54	0.57	0.55
graduate-jobs (125)	0.00	0.00	0.00
hr-jobs (258)	0.00	0.00	0.00
healthcare-nursing-jobs (5775)	0.86	0.93	0.89
hospitality-catering-jobs (2944)	0.82	0.88	0.84
it-jobs (1037)	0.62	0.76	0.68
legal-jobs (221)	0.00	0.00	0.00
logistics-warehouse-jobs (3877)	0.76	0.88	0.81
maintenance-jobs (910)	0.46	0.36	0.41
manufacturing-jobs (1629)	0.55	0.64	0.60
other-general-jobs (5122)	0.57	0.44	0.50
pr-advertising-marketing-jobs (452)	0.63	0.29	0.40
property-jobs (167)	0.00	0.00	0.00
retail-jobs (2196)	0.86	0.81	0.84
sales-jobs (1509)	0.70	0.82	0.76
scientific-qa-jobs (429)	0.63	0.51	0.57
security-protective-services-jobs (910)	0.92	0.92	0.92
social-work-jobs (761)	0.58	0.47	0.52
teaching-jobs (1203)	0.71	0.81	0.76
trade-construction-jobs (6912)	0.87	0.93	0.90
travel-jobs (151)	0.00	0.00	0.00

Figure 39: Classification Report for FNN with custom FastText embeddings (skipgram) with lower case descriptions

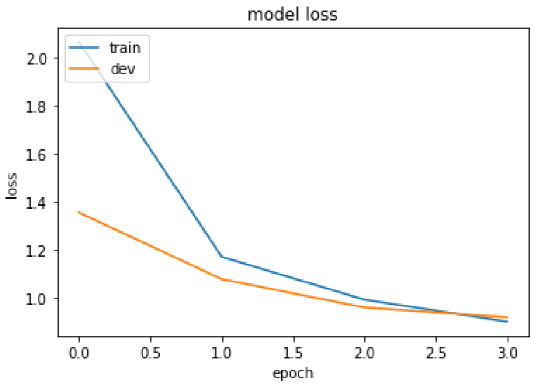
12) Convolutional Neural Network trained on Google's word2vec (Glove) embeddings with lower case descriptions

The Convolutional Neural Network on sequences of words and on Google's word2vec (Glove) embeddings of lower case descriptions achieved a training accuracy of 72.58% and a testing accuracy of 73.5%. The macro average Precision, Recall and F1-Score were 62%, 49% and 50% respectively. The weighted average Precision, Recall and F1-Score were 73%, 73% and 72% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 42. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 16 below:

	Precision	**Recall**	**F1-Score**
Macro average	62%	49%	50%
Weighted average	72%	73%	71%

Table 16: Average Metrics for CNN on Glove embeddings with lower case descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 40 and 41 respectively.



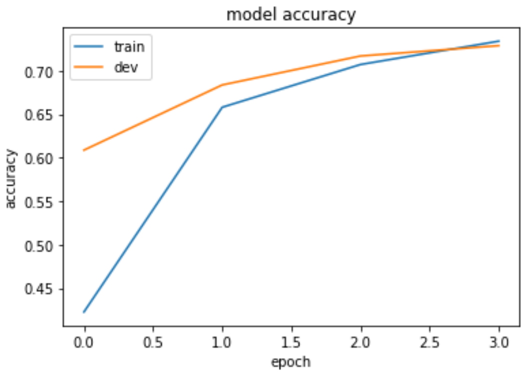


Figure 40: Loss per training epoch for CNN with Glove embeddings with lower case descriptions

Figure 41: Accuracy per training epoch for CNN with with Glove embeddings with lower case descriptions

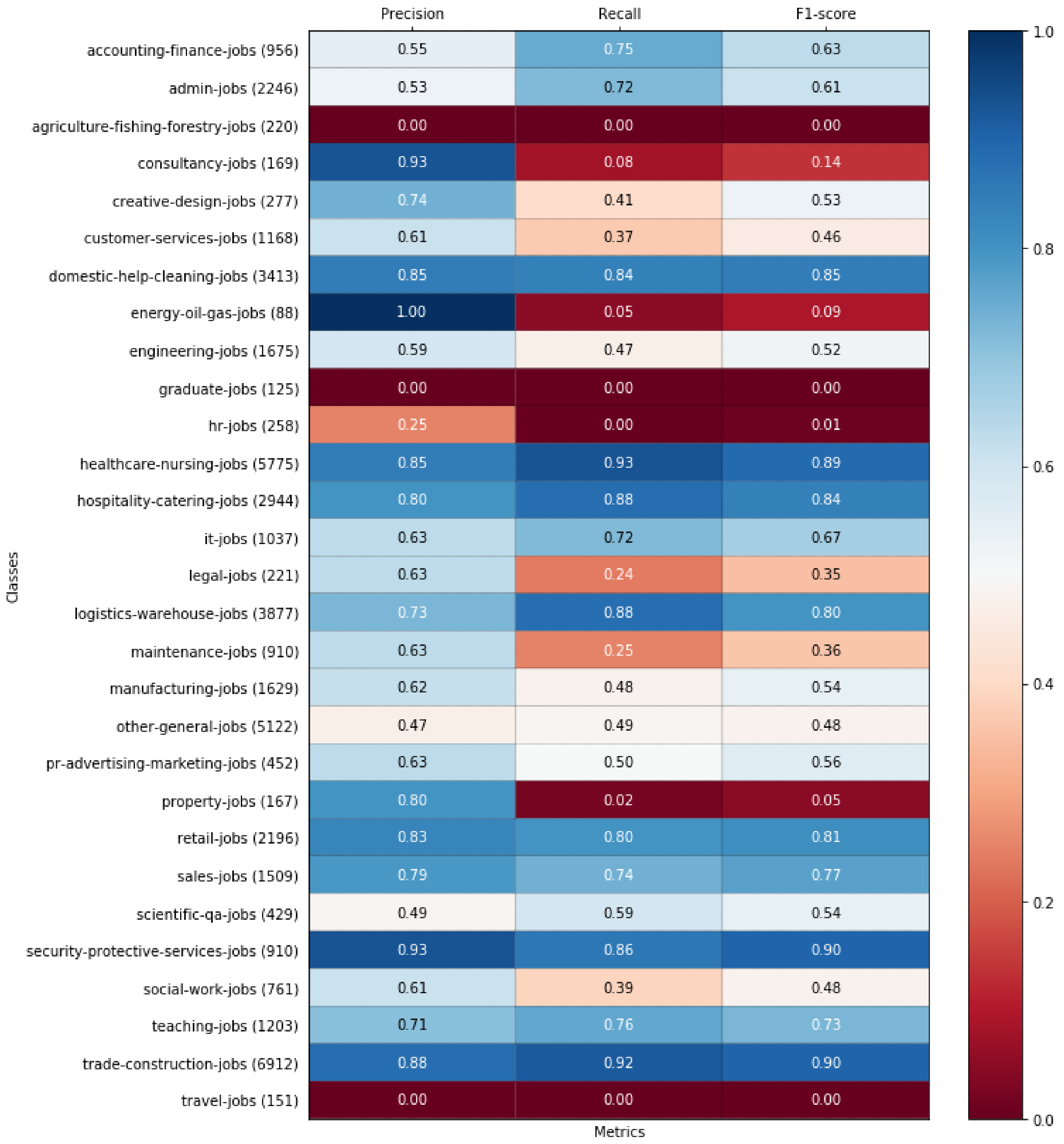


Figure 42: Classification Report for CNN with Glove embeddings with lower case descriptions

13) Convolutional Neural Network trained on custom word2vec embeddings (with skipgram algorithm) with lower case descriptions

The Convolutional Neural Network on sequences of words and on custom word2vec embeddings (trained with skipgram) of lower case descriptions achieved a training accuracy of 76. 8% and a testing accuracy of 76.17%. The macro average Precision, Recall and F1-Score were 65%, 54% and 56% respectively. The weighted average Precision, Recall and F1-Score were 75%, 76% and 77% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 45. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 17 below:

	Precision	Recall	F1-Score
Macro average	65%	54%	56%
Weighted average	75%	76%	77%

Table 17: Average Metrics for CNN on custom word2vec embeddings with lower case descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 43 and 44 respectively.

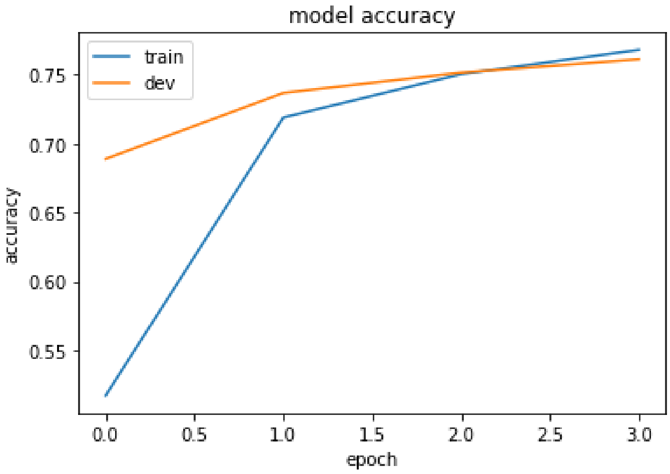


Figure 43: Loss per training epoch for CNN custom word2vec embeddings (skipgram) with lower case descriptions

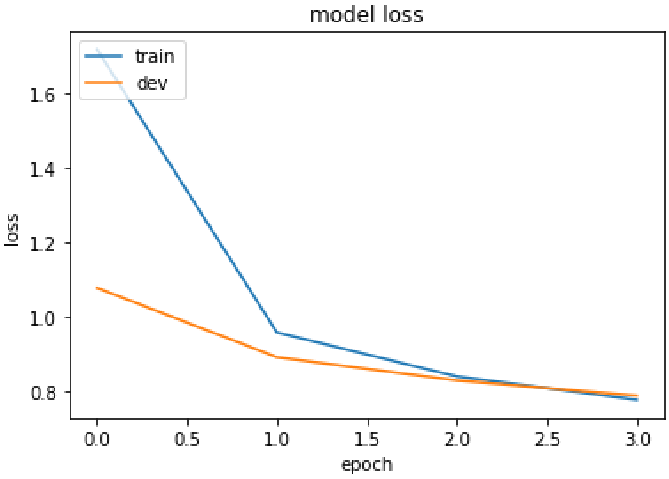


Figure 44: Accuracy per training epoch for CNN custom word2vec embeddings (skipgram) with lower case descriptions

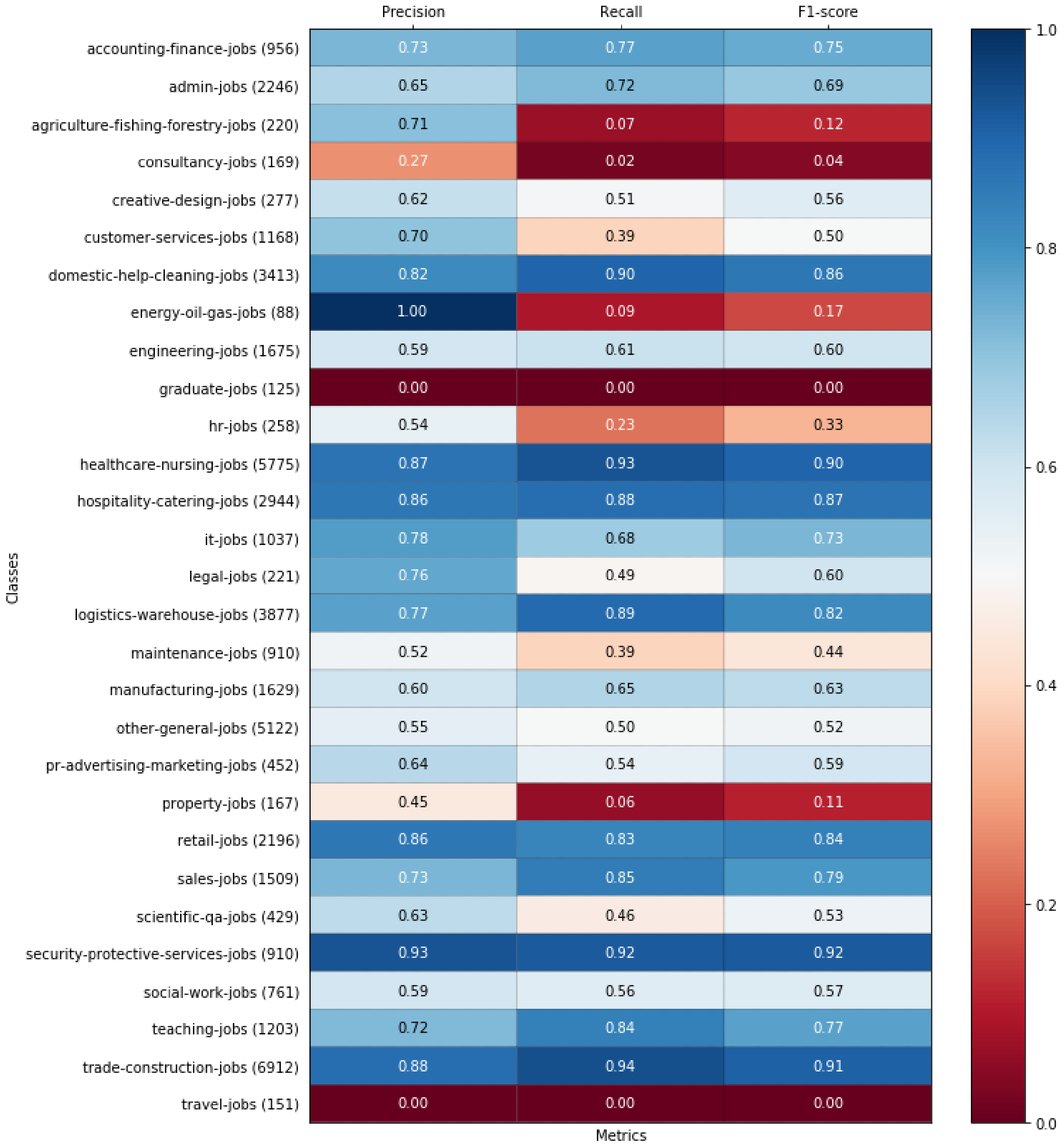


Figure 45: Classification Report for CNN with custom word2vec embeddings (with skipgram) with lower case descriptions

14) Convolutional Neural Network trained on custom FastText embeddings (with skipgram algorithm) with lower case descriptions

The Convolutional Neural Network on sequences of words and on custom FastText embeddings (trained with skipgram) of lower case descriptions achieved a training accuracy of 76.27% and a testing accuracy of 75.65%. The macro average Precision, Recall and F1-Score were 65%, 53% and 54% respectively. The weighted average Precision, Recall and F1-Score were 75%, 76% and 74% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 48. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 18 below:

	Precision	Recall	F1-Score
Macro average	65%	53%	54%
Weighted average	75%	76%	74%

Table 18: Average Metrics for CNN on custom word2vec embeddings with lower case descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 46 and 47 respectively.

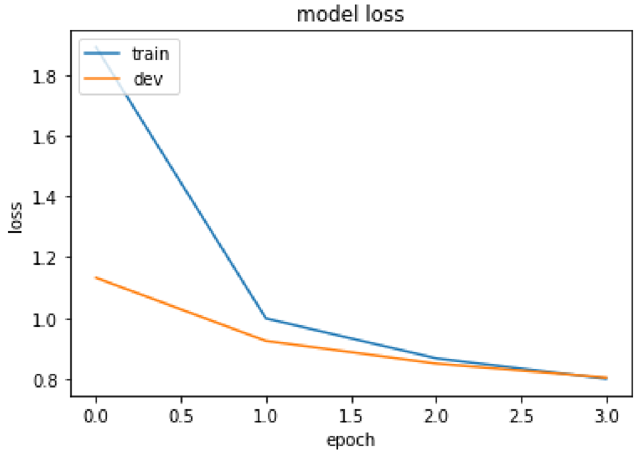


Figure 46: Loss per training epoch for CNN with custom FastText embeddings (skipgram) with lower case descriptions

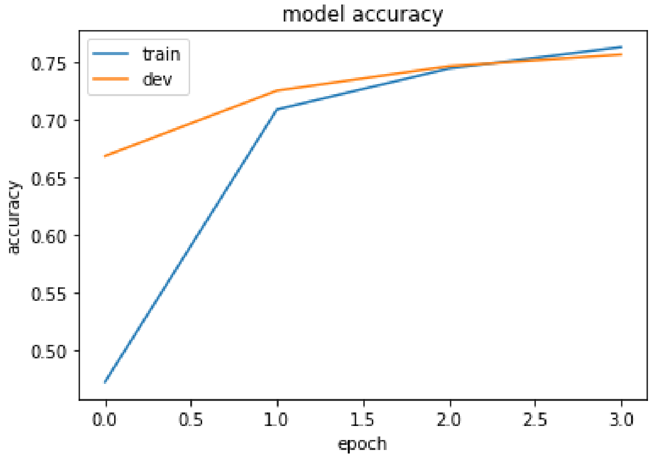


Figure 47: Accuracy per training epoch for CNN with custom FastText word2vec embeddings (skipgram) with lower case descriptions

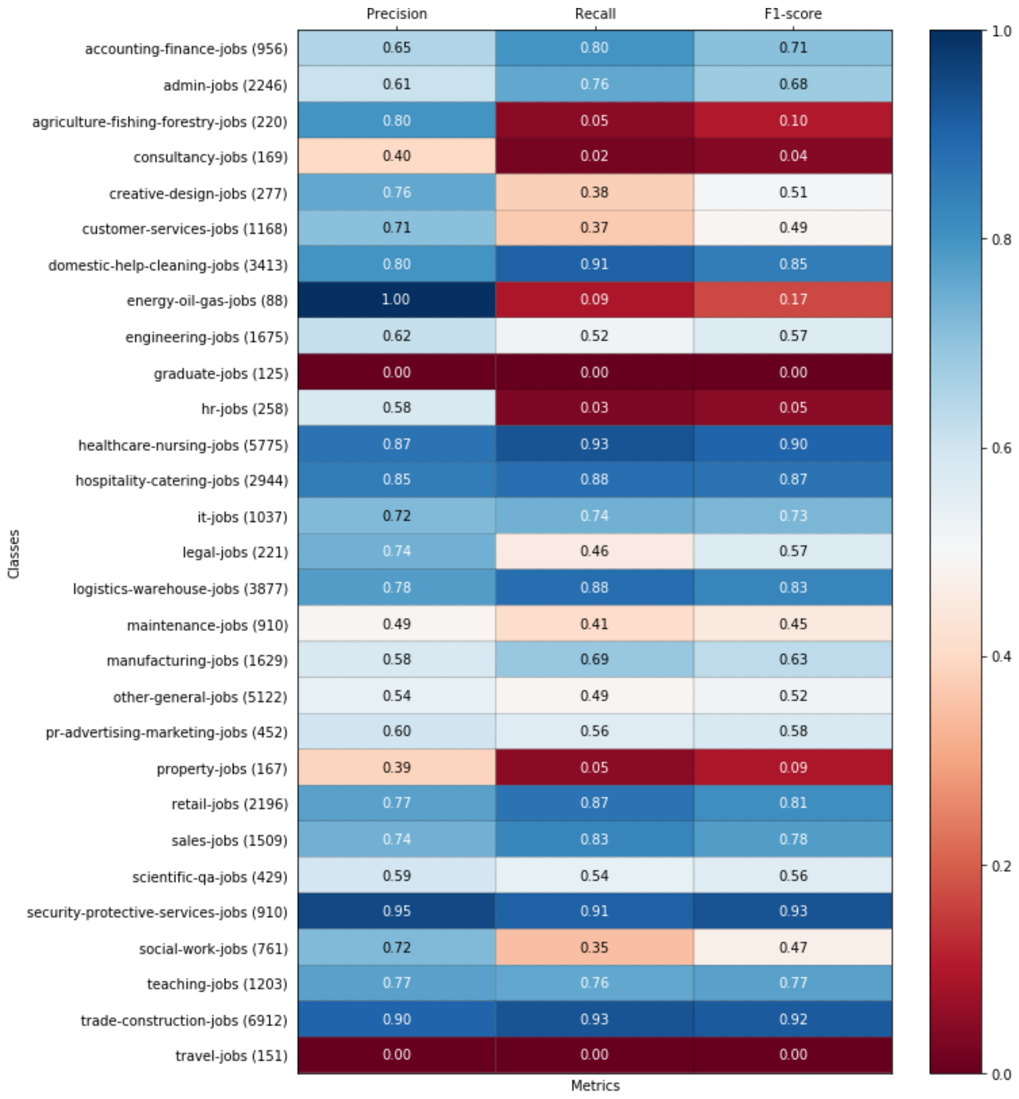


Figure 48: Classification Report for CNN with custom FastText embeddings (with skipgram) with lower case descriptions

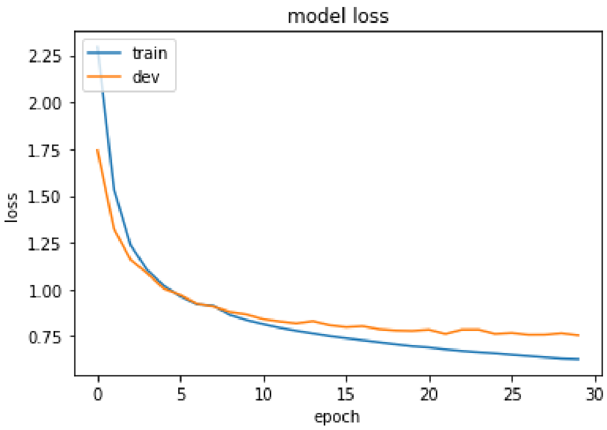
15) Recurrent Neural Network trained on Google's word2vec (Glove) embeddings with lower case descriptions

The Recurrent Neural Network on sequences of words and on Glove embeddings with lower case descriptions achieved a training accuracy of 81.08% and a testing accuracy of 78.25%. The macro average Precision, Recall and F1-Score were 68%, 61% and 63% respectively. The weighted average Precision, Recall and F1-Score were 77%, 78% and 77% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 51. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 19 below:

	Precision	**Recall**	**F1-Score**
Macro average	68%	61%	63%
Weighted average	77%	78%	77%

Table 19: Average Metrics for RNN on Glove embeddings with lower case descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 49 and 50 respectively.



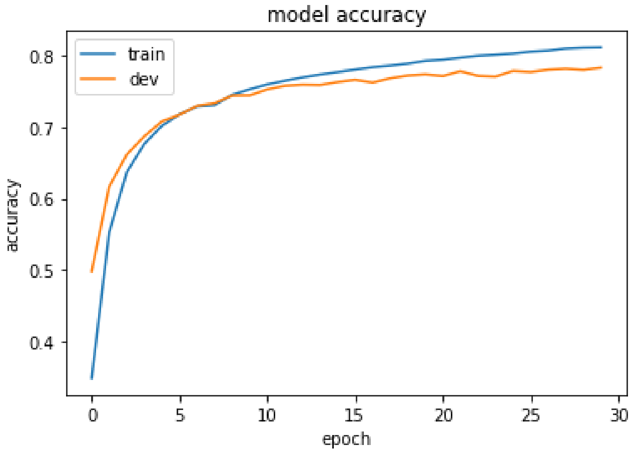


Figure 49: Loss per training epoch for RNN on Glove embeddings with lower case descriptions

Figure 50: Accuracy per training epoch for RNN on Glove embeddings with lower case descriptions

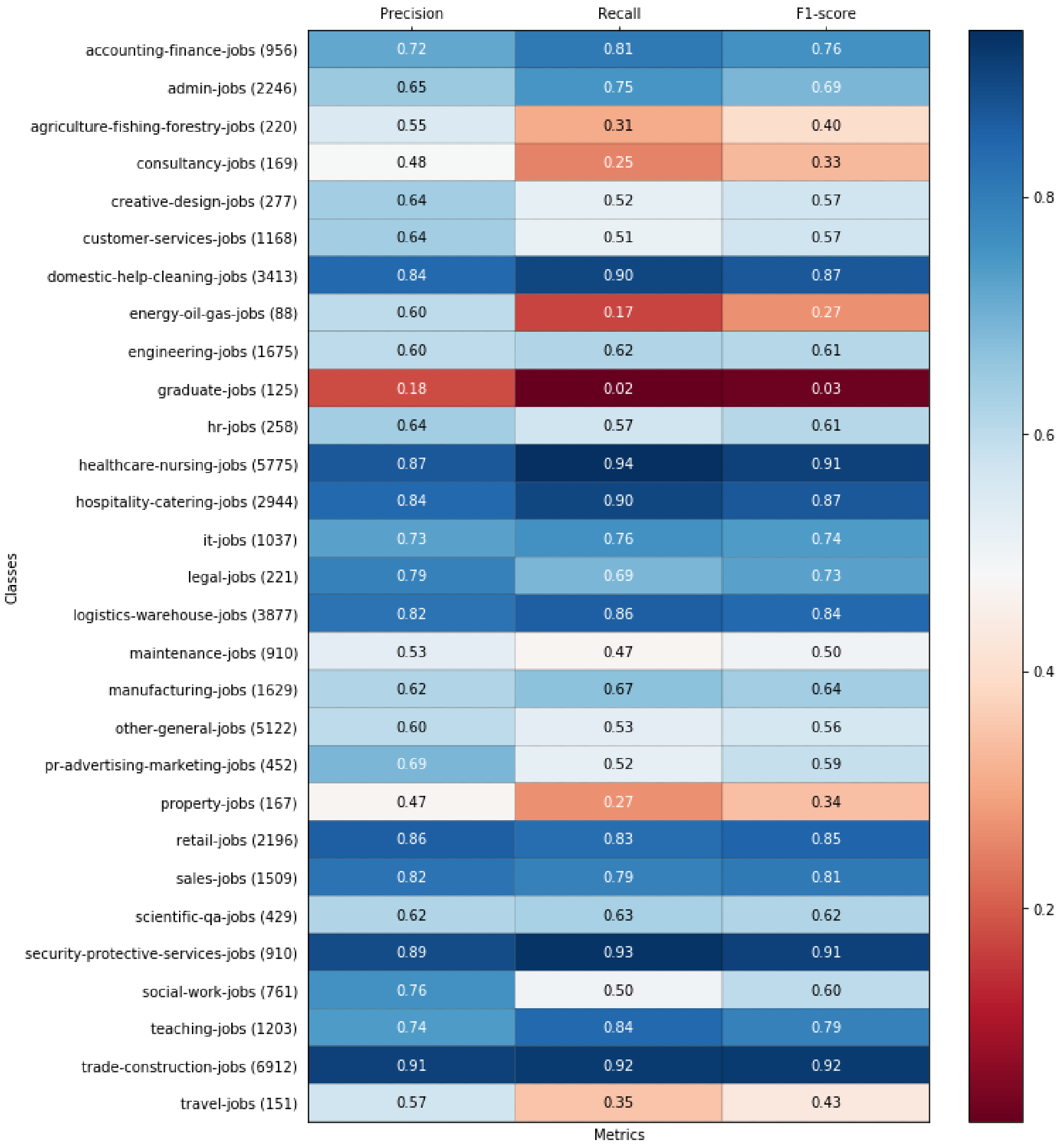


Figure 51: Classification Report for RNN on Glove embeddings with lower case descriptions

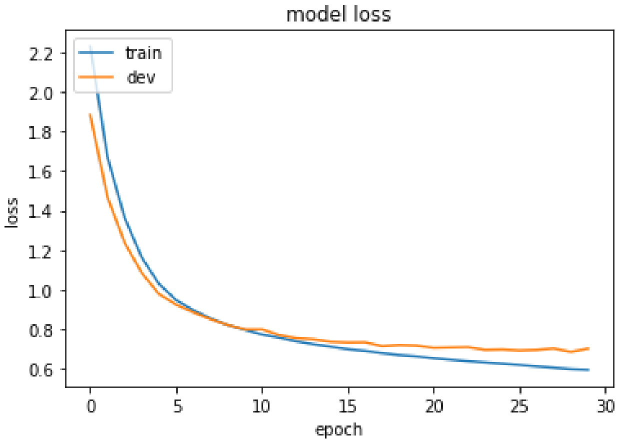
16) Recurrent Neural Network trained on custom word2vec embeddings (with skipgram algorithm) with lower case descriptions

The Recurrent Neural Network on sequences of words and on custom word2vec embeddings (trained with skipgram) with lower case descriptions achieved a training accuracy of 81.74% and a testing accuracy of 79.38%. The macro average Precision, Recall and F1-Score were 71%, 63% and 65% respectively. The weighted average Precision, Recall and F1-Score were 78%, 79% and 78% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 54. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 20 below:

	Precision	**Recall**	**F1-Score**
Macro average	71%	63%	65%
Weighted average	78%	79%	78%

Table 20: Average Metrics for RNN on custom word2vec embeddings with lower case descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 52 and 53 respectively.



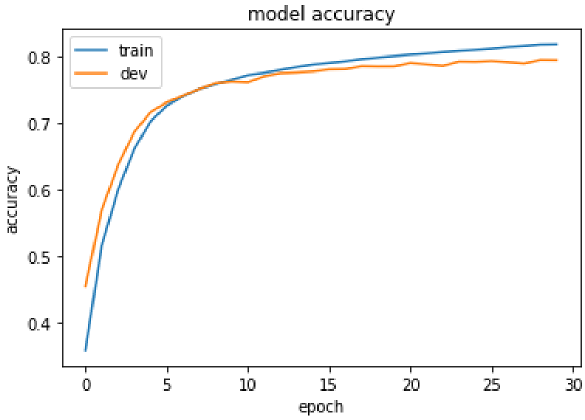


Figure 52: Loss per training epoch for RNN on custom word2vec embeddings with lower case descriptions

Figure 53: Accuracy per training epoch for RNN on custom word2vec embeddings with lower case descriptions

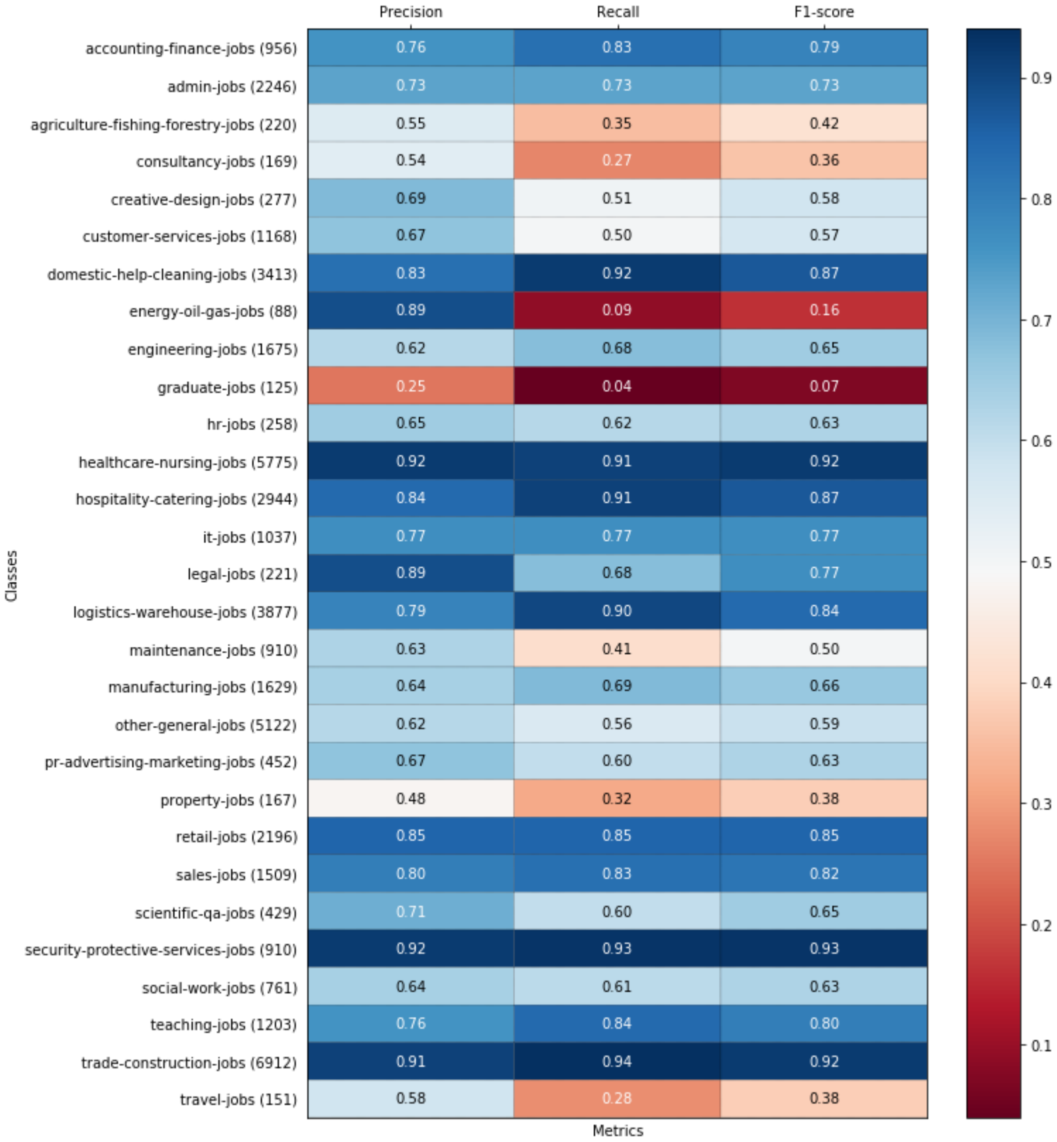


Figure 54: Classification Report for RNN on custom word2vec embeddings with lower case descriptions

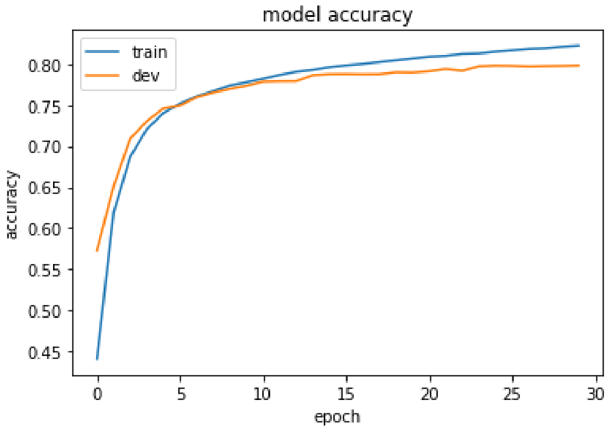
17) Recurrent Neural Network trained on custom FastText embeddings (with skipgram algorithm) with lower case descriptions

The Recurrent Neural Network on sequences of words and on custom FastText embeddings (trained with skipgram) with lower case descriptions achieved a training accuracy of 82.22% and a testing accuracy of 79.22%. The macro average Precision, Recall and F1-Score were 71%, 63% and 65% respectively. The weighted average Precision, Recall and F1-Score were 79%, 79% and 79% respectively. The Precision, Recall and F1-Score of each class is highlighted in Figure 57. The confusion matrix is available at the Appendix. The average F1-Score, Precision and Recall of the model are presented in Table 21 below:

	Precision	**Recall**	**F1-Score**
Macro average	71%	63%	65%
Weighted average	79%	79%	79%

Table 21: Average Metrics for RNN on custom FastText embeddings with lower case descriptions

The model loss (train and test) and model accuracy (train and test) with respect to the training epochs are presented in Figures 55 and 56 respectively.



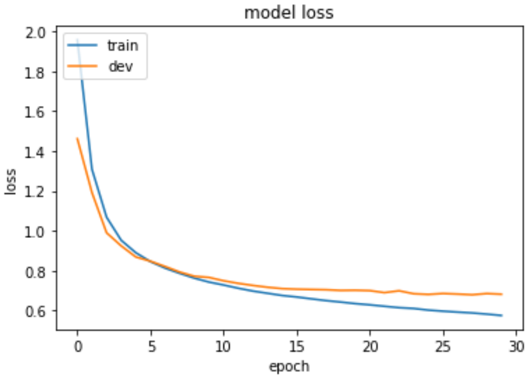


Figure 55: Loss per training epoch for RNN on custom FastText embeddings with lower case descriptions

Figure 56: Accuracy per training epoch for RNN on custom FastText embeddings with lower case descriptions

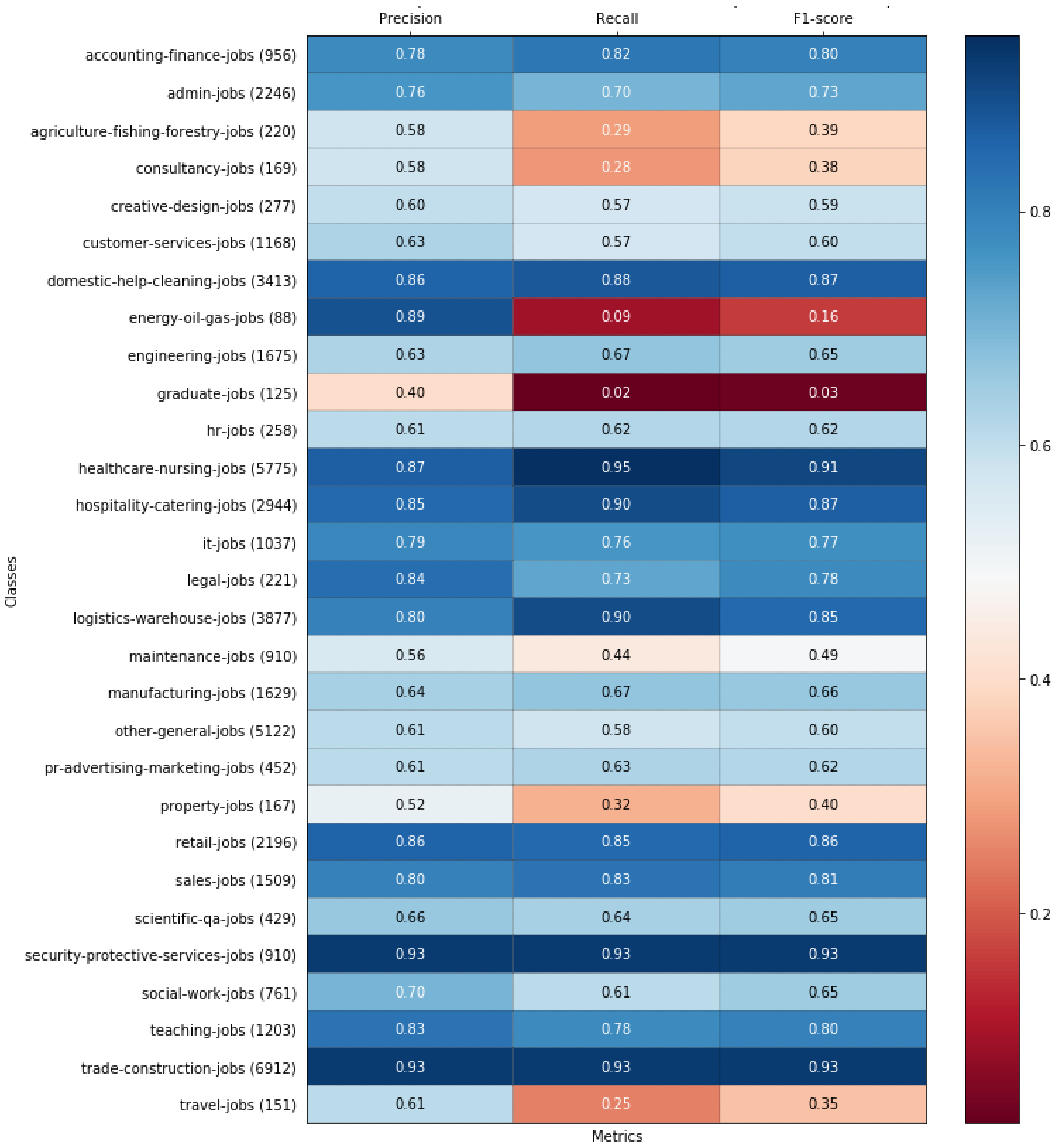


Figure 57: Classification Report for RNN on custom FastText embeddings withc

6.2 Model comparison

A summarized model-wise comparison of achieved metrics is presented in Figure 58. The model that achieved the highest weighted average F1-Score was the Feedforward Neural Network trained on BoW (TF-IDF) representations of lowercased and stemmed job descriptions. Spefically, this model achieved a weighted average Precision of 80%, a weighted average Recall of 81%, a weighted average F1-Score of 80% and a total test accuracy of 80.5%. The other Feedforward Neural Network trained on BoW (on just lowercased job description) had similar results: weighted average Precision of 80%, a weighted average Recall of 81%, a weighted average F1-Score of 80% and a total test accuracy of 80.37%. In this case, the stemming of job descriptions does not seem to give significantly better results. In all other cases of BoW representations, this is also obvious: If we examine each model family (MNB, DT, SVC) separately, we conclude that training on lowercased and training on lowercased and stemmed job descriptions results in similar metrics. For instance, the SVC on BoW representation of lowercased descriptions and the SVC on BoW representation of lowercased and stemmed description achieved exactly the same test accuracy and weighted average F1-Score.

The results show that Deep Learning models outperform all kinds of conventional Machine Learning approaches. Specifically, for BoW (TF-IDF) representation of lowercased and stemmed job descriptions, the trained Feedforward Neural Network achieved weighted average Precision, Recall and F1-Score were 80%, 81% and 80% respectively outperforming all trained Support Vector Classifiers, Decision Trees and Multinomial naïve Bayes models. Regarding strictly the conventional Machine Learning approaches, the Support Vector Classifier with linear kernel trained on BoW representation of lower case descriptions achieved a training accuracy of 79.5% and a testing accuracy of 77.3% outperforming all other Decision Trees and Multinomial Naïve Bayes models. The Decision Trees had the lowest metrics among all Machine Learning models (and among all trained models).

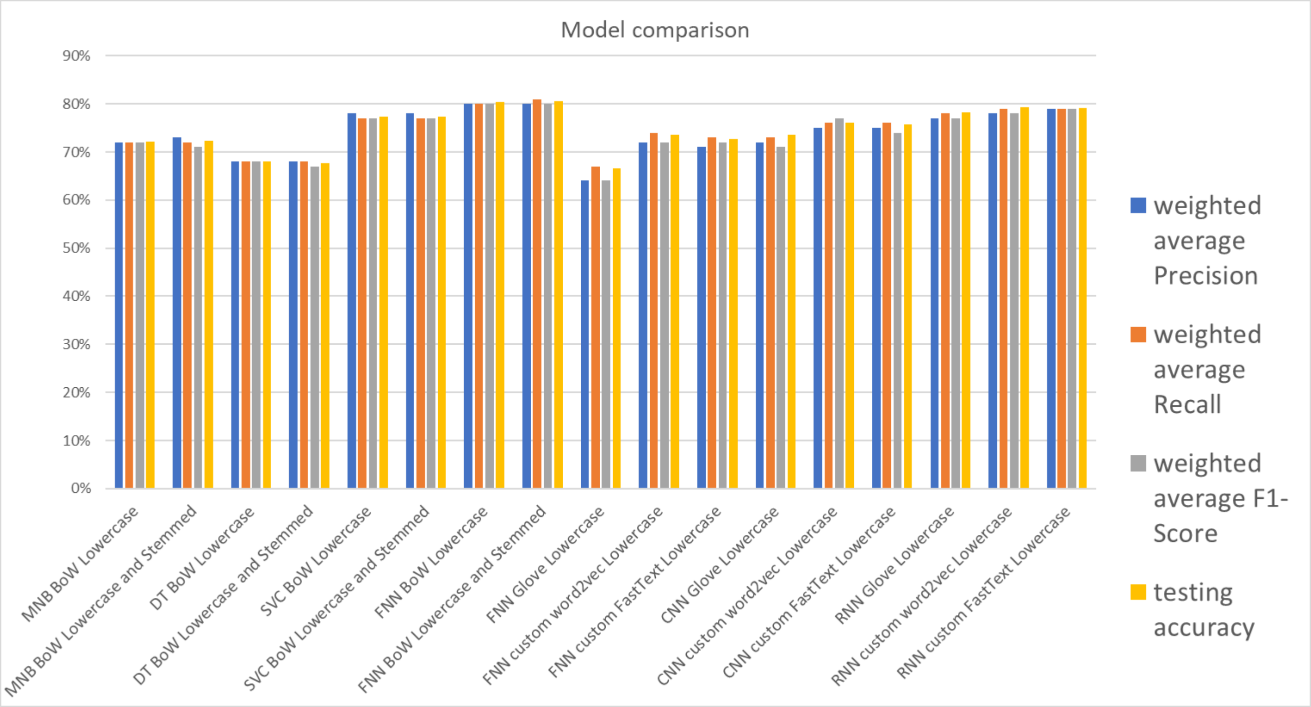


Figure 58: Model comparison with respect to metrics (weighted)

If we examine just the Feedforward Neural Networks and the 5 different input representations we fed into them, then we conclude that FFNs trained on any BoW representation outperform all kinds of approaches with FNNs and embeddings. However, we cannot generalize this into the assumption that BoW representations give better results compared to embeddings and sequences of words. This is due to the fact that all other kinds of NNs (RNNs and CNNs) when trained on sequences of words with any kind of embedding layer outperformed all types of conventional Machine Learning model trained on BoW input (MNB, DT and SVC). It seems that the model selection is of higher importance in our sample of trained models and their metrics. Specifically, all Deep Learning models rank higher in terms of metrics compared to conventional Machine Learning algorithms.

Investigating strictly just the different embeddings used, we conclude that custom trained word2vec embeddings give better testing accuracies than any other embedding (custom trained FastText and pretrained Glove) for each separate neural network architecture (feedforward, convolutional and recurrent). As presented in Figure 59, for all types of neural network architectures, the custom word2vec embedding always achieves higher testing accuracies. Additionally, pre-trained Glove always gives the lowest testing accuracy. However, for Recurrent Neural Networks, the Glove embedding achieves similar testing accuracy with the other two embedding types. Finally, the custom FastText embedding achieves close testing accuracy (and weighted average metrics) with custom word2vec embeddings. Among Neural Networks trained on embeddings, the Recurrent Neural Networks trained on lowercased job descriptions and custom word2vec embedding layer is the best performing model as it achieves a testing accuracy of 79.38% and weighted average Precision, Recall and F1-Score were 78%, 79% and 78% respectively.

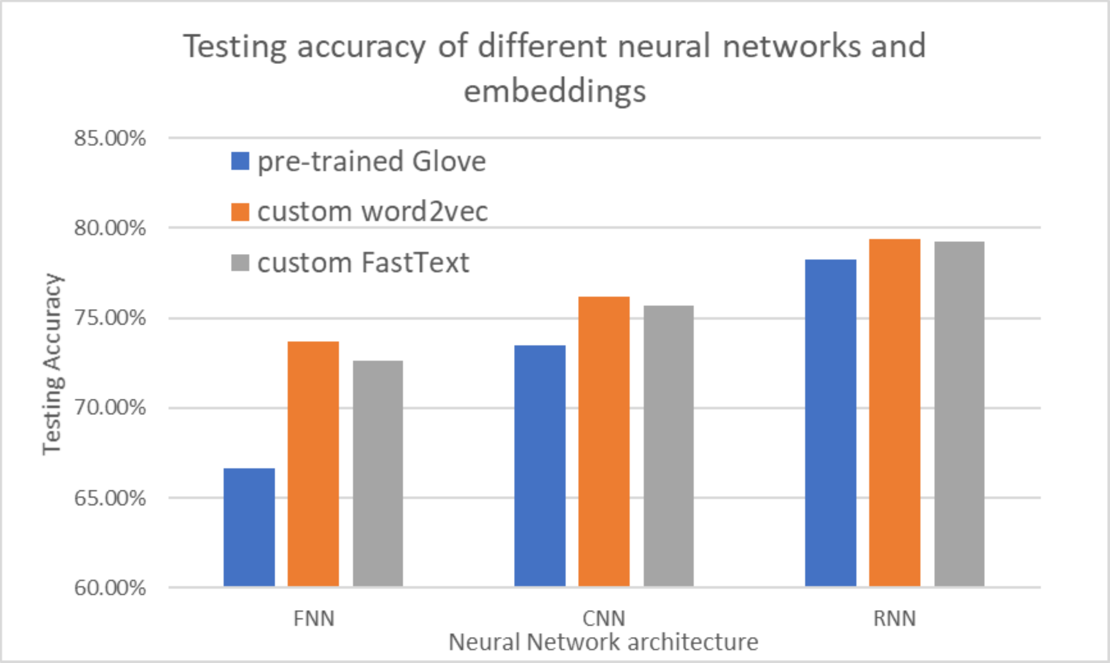


Figure 59: Testing accuracy per embedding type

The most frequent and dominant misclassifications per class among all models are presented in Table 22 below:

Actual	Predicted
Admin	Accounting, Customer Services, Human Resources, Other General, Sales
Engineering	Trade and Construction, Manufacturing, Maintenance, Other General
Domestic Help and Cleaning	Other General
Healthcare	Social Work, Other General
Hospitality and Catering	Other and General
Logistics and Warehousing	Other and General
Manufacturing	Trade and Construction

Teaching	Other and General
Manufacturing	Engineering

Table 22: Most common misclassifications among the models

Some of these false classifications can be interpreted by the similar content of the categories. For instance, healthcare and social work jobs as well as admin and accounting jobs can have common job descriptions. This is due to use of same hint words, that exist in both sectors. The rates of misclassifications between the described categories seem to decrease when using Deep Learning models rather than conventional Machine Learning models.

In addition, some categories have zero Precision and Recall (and therefore zero F1) when predicted by some conventional Machine Learning models. This is due to the significant lower training data available for these categories. The categories of Agriculture/Fishing/Forestry, Property, Graduate and Travel are such categories: Multinomial Naïve Bayes preformed poorly on theses classes. Deep Learning models seem to balance these phenomena, as minority classes have higher True Positive rates. Specifically, RNNs achieve low recall and precisions for these 4 classes rather than zeros.

Furthermore, there are some categories which show good Precisions and significantly low Recall such as Energy-Oil-Gas and Consultancy. This means that there are plenty of False Positives, which can be due to some kind of overfitting to these classes. These phenomena, again, seem to be more intense in the cases of SVC, Multinomial Naïve Bayes and Decision Trees. Finally, there are some minority classes such as Healthcare and Legal jobs that have good metrics despite the significantly low available data.

7 Implementation Issues

All described Deep Learning and Machine Learning methodologies were implemented using Python 3.7.1 and the Keras open-source library (version 2.2.4-tf) running on top of the Tensorflow Deep Learning framework (version 2.0.0) [63, 64]. The infrastructure used was an ASUS ROG GL704GV Laptop with 16GB of Random-access memory (RAM), a solid-state drive (SSD) of 256GB capacity, a solid-state drive (SSD) of 1000GB capacity, a graphics card NVIDIA RTX 2060 (laptop version) and an Intel Core i7-8750H Processor. Both SSDs were configured to act as extra RAM space dynamically in case of full RAM. The essential python libraries used for the purposes of data preprocessing, training and inferencing the models and processing and visualizing results are presented in Table 22.

NLTK	Seaborn
Pandas	Matplotlib
Sklearn	Plotly
Imblearn	Numpy

Table 23: Python libraries used

8 Conclusion and Future Work

Our main results show that Deep Learning models outperform all kinds of conventional Machine Learning approaches such as Support Vector Classifiers, Multinomial Naïve Bayes and Decision Trees. In addition, training custom word2vec embeddings helps achieve higher accuracy metrics compared to using pretrained embeddings such as Glove 100. However, the model selection (choosing a Deep Learning model against a conventional Machine Learning model) is of higher impact towards better metrics than using embeddings and sequences of words. The model that achieved the highest weighted average F1-Score (80%) and the highest testing accuracy (80.5%) was the Feedforward Neural Network trained on Bag of Words (TF-IDF) representations of lowercased and stemmed job descriptions. These metrics are significantly higher than the metrics achieved by the current implementation of job categorization. Specifically, this model achieved a weighted average Precision of 80%, a weighted average Recall of 81%.

Deep Learning models on custom trained embeddings seem to be working good in predominant classes. There are plenty of misclassification among related job categories, which are more often in the predictions of the conventional Machine Learning models. In addition, there are some categories which show good Precisions and significantly low Recall due to the imbalance of the dataset and the high context similarity among specific categories. Furthermore, all types of models perform poorly on specific minority classes, specifically Agriculture/Fishing/Forestry, Property, Graduate and Travel job categories. Finally, there is high imbalance between achieved Recall and Precision among the classes.

The first action to achieve more balanced precision and recall results among all classes is to increase the training data in currently minority classes. Experimenting with undersampling or oversampling as well as data augmentation techniques such as synthetic data generation could be also promising towards increasing metrics of minority classes. It could also be beneficial if we get rid of some categories, for instance by being merged into some other related category.

Another method to achieve better results is to use some kind of ensemble method: Some models seem to be better in predicting specific classes, while being bad at predicting other. We could experiment with creating model layers in a bigger architecture. For instance, we could have a first layer of trained models, which can only predict two classes and an “other” class. If the prediction being made by this first layer is the “other” class, then we proceed to the next layer, which is built upon another model. This kind of “prediction filtering” can help train separate and less complex models.

Using probabilistic algorithms (with probabilistic output) could also be beneficial. For instance, we could use these outputs in order to create different confidence levels per class prediction. An example could be to fine tune all these thresholds with respect to achieved metrics and accept each classification only if the probability of prediction is higher than the tuned class specific threshold. In the other case, we would automatically assign to the instance an “unknown” category. This could help decrease the False Positive rates, as we would accept only strong predictions. This method would, however, require tuning of 29 probability thresholds which increases the computational cost of training and development.

Finally, as the main contextual information for job adverts by nature seems to be the skillset required for each different vacancy, one could try to extract the skills from each document and use them to train a model instead of the whole document. This way, there is a significant reduction in the dimensionality of the dataset and a possibility that the classes would become more distinct, as in theory, job adverts from different categories would consist of different required skillsets too.

9 BIBLIOGRAPHY

[1] Adzuna website, available at: https://www.adzuna.co.uk/, last accessed on January 2021

[2] National Health System, available at: https://www.nhs.uk/, last accessed on January, 2021

[3] Zhang, X., Zhao, J., & LeCun, Y. (2015). Character-level convolutional networks for text classification. arXiv preprint arXiv:1509.01626.

[4] Aggarwal, C. C., & Zhai, C. (2012). A survey of text classification algorithms. In Mining text data (pp. 163-222). Springer, Boston, MA.

[5] Li, L., Wang, D., Li, T., Knox, D., & Padmanabhan, B. (2011, July). Scene: a scalable two-stage personalized news recommendation system. In Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval (pp. 125-134).

[6] Lee, K., Palsetia, D., Narayanan, R., Patwary, M. M. A., Agrawal, A., & Choudhary, A. (2011, December). Twitter trending topic classification. In 2011 IEEE 11th International Conference on Data Mining Workshops (pp. 251-258). IEEE.

[7] Quercia, D., Askham, H., & Crowcroft, J. (2012, June). Tweetlda: supervised topic classification and link prediction in twitter. In Proceedings of the 4th Annual ACM Web Science Conference (pp. 247-250).

[8] Hillard, D., Purpura, S., & Wilkerson, J. (2008). Computer-assisted topic classification for mixed-methods social science research. Journal of Information Technology & Politics, 4(4), 31-46.

[9] Wiltshire Jr, J. S., Morelock, J. T., Humphrey, T. L., Lu, X. A., Peck, J. M., & Ahmed, S. (2002). U.S. Patent No. 6,502,081. Washington, DC: U.S. Patent and Trademark Office.

[10] Kumar, M., & Rangan, V. (2011). U.S. Patent No. 7,899,871. Washington, DC: U.S. Patent and Trademark Office.

[11] Pang, B., Lee, L., & Vaithyanathan, S. (2002). Thumbs up? Sentiment classification using machine learning techniques. arXiv preprint cs/0205070.

[12] Wang, S. I., & Manning, C. D. (2012, July). Baselines and bigrams: Simple, good sentiment and topic classification. In Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers) (pp. 90-94).

[13] Maas, A., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011, June). Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies* (pp. 142-150).

[14] Pang, B., & Lee, L. (2004). A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. *arXiv preprint cs/0409058*.

[15] Sahami, M., Dumais, S., Heckerman, D., & Horvitz, E. (1998, July). A Bayesian approach to filtering junk e-mail. In *Learning for Text Categorization: Papers from the 1998 workshop* (Vol. 62, pp. 98-105).

[16] Rathod, S. B., & Pattewar, T. M. (2015, April). Content based spam detection in email using Bayesian classifier. In *2015 International Conference on Communications and Signal Processing (ICCSP)* (pp. 1257-1261). IEEE.

[17] Sasaki, M., & Shinnou, H. (2005, November). Spam detection using text clustering. In *2005 International Conference on Cyberworlds (CW'05)* (pp. 4-pp). IEEE.

[18] Ren, Y., & Ji, D. (2017). Neural networks for deceptive opinion spam detection: An empirical study. *Information Sciences*, *385*, 213-224.

[19] Jindal, N., & Liu, B. (2007, May). Review spam detection. In *Proceedings of the 16th international conference on World Wide Web* (pp. 1189-1190).

[20] Saidani, N., Adi, K., & Allili, M. S. (2019, November). Semantic representation based on deep learning for spam detection. In *International Symposium on Foundations and Practice of Security* (pp. 72-81). Springer, Cham.

[21] Zheng, R., Li, J., Chen, H., & Huang, Z. (2006). A framework for authorship identification of online messages: Writing-style features and classification techniques. *Journal of the American society for information science and technology*, *57*(3), 378-393.

[22] Rajaraman, A., & Ullman, J. D. (2011). *Mining of massive datasets*. Cambridge University Press.

[23] Stuart, A., Arnold, S., Ord, J. K., O'Hagan, A., & Forster, J. (1994). *Kendall's advanced theory of statistics*. Wiley.

[24] Jurafsky, Dan, and C. Manning. "Text Classification and Naïve Bayes." *Available at http://www. stanford. edu/class/cs124/lec/naivebayes.pdf (last accede on January 2021)* (2015).

[25] https://scikit-learn.org/stable/modules/naive_bayes.html

[26] Kibriya, A. M., Frank, E., Pfahringer, B., & Holmes, G. (2004, December). Multinomial naive bayes for text categorization revisited. In *Australasian Joint Conference on Artificial Intelligence* (pp. 488-499). Springer, Berlin, Heidelberg.

[27] Frank, E., & Bouckaert, R. R. (2006, September). Naive bayes for text classification with unbalanced classes. In *European Conference on Principles of Data Mining and Knowledge Discovery* (pp. 503-510). Springer, Berlin, Heidelberg.

[28] Bahety, A. (2014). Extension and evaluation of id3–decision tree algorithm. *Entropy (S)*, *2*(1), 1-8.

[29] Hssina, B., Merbouha, A., Ezzikouri, H., & Erritali, M. (2014). A comparative study of decision tree ID3 and C4. 5. *International Journal of Advanced Computer Science and Applications*, *4*(2), 13-19.

[30] Wang, Z. Q., Sun, X., Zhang, D. X., & Li, X. (2006, August). An optimal SVM-based text classification algorithm. In *2006 International Conference on Machine Learning and Cybernetics* (pp. 1378-1381). IEEE.

[31] Moraes, R., Valiati, J. F., & Neto, W. P. G. (2013). Document-level sentiment classification: An empirical comparison between SVM and ANN. *Expert Systems with Applications*, *40*(2), 621-633.

[32] Joachims, T. (2002). *Learning to classify text using support vector machines* (Vol. 668). Springer Science & Business Media.

[33] Sun, A., Lim, E. P., & Liu, Y. (2009). On strategies for imbalanced text classification using SVM: A comparative study. *Decision Support Systems*, *48*(1), 191-201.

[34] Wang, Y., Zhou, Z., Jin, S., Liu, D., & Lu, M. (2017, October). Comparisons and selections of features and classifiers for short text classification. In *IOP Conference Series: Materials Science and Engineering* (Vol. 261, No. 1, p. 012018). IOP Publishing.

[35] Joachims, T. (1998, April). Text categorization with support vector machines: Learning with many relevant features. In *European conference on machine learning* (pp. 137-142). Springer, Berlin, Heidelberg.

[36] Kivinen, J., Warmuth, M. K., & Auer, P. (1997). The Perceptron algorithm versus Winnow: linear versus logarithmic mistake bounds when few input variables are relevant. *Artificial Intelligence*, *97*(1-2), 325-343.

[37] Schölkopf, B., Tsuda, K., & Vert, J. P. (2004). *Kernel methods in computational biology*. MIT press.

[38] Lin, H. T., & Lin, C. J. (2003). A study on sigmoid kernels for SVM and the training of non-PSD kernels by SMO-type methods. *submitted to Neural Computation*, *3*(1-32), 16.

[39] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[40] Svozil, D., Kvasnicka, V., & Pospichal, J. (1997). Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, *39*(1), 43-62.

[41] Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1, No. 2). Cambridge: MIT press.

[42] Wang, G., Li, C., Wang, W., Zhang, Y., Shen, D., Zhang, X., ... & Carin, L. (2018). Joint embedding of words and labels for text classification. *arXiv preprint arXiv:1805.04174*.

[43] Stein, R. A., Jaques, P. A., & Valiati, J. F. (2019). An analysis of hierarchical text classification using word embeddings. *Information Sciences*, *471*, 216-232.

[44] Joulin, A., Grave, E., Bojanowski, P., & Mikolov, T. (2016). Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*.

[45] Lai, S., Xu, L., Liu, K., & Zhao, J. (2015, February). Recurrent convolutional neural networks for text classification. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 29, No. 1).

[46] Rao, A., & Spasojevic, N. (2016). Actionable and political text classification using word embeddings and lstm. *arXiv preprint arXiv:1607.02501*.

[47] Hughes, M., Li, I., Kotoulas, S., & Suzumura, T. (2017). Medical text classification using convolutional neural networks. *Stud Health Technol Inform*, *235*, 246-50.

[48] https://nlp.stanford.edu/projects/glove/

[49] Pennington, J., Socher, R., & Manning, C. D. (2014, October). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532-1543).

[50] https://radimrehurek.com/gensim/

[51] Arevian, G. (2007, November). Recurrent neural networks for robust real-world text classification. In *IEEE/WIC/ACM International Conference on Web Intelligence (WI'07)* (pp. 326-329). IEEE.

[52] Nowak, J., Taspinar, A., & Scherer, R. (2017, June). LSTM recurrent neural networks for short text and sentiment classification. In *International Conference on Artificial Intelligence and Soft Computing* (pp. 553-562). Springer, Cham.

[53] Rao, G., Huang, W., Feng, Z., & Cong, Q. (2018). LSTM with sentence representations for document-level sentiment classification. *Neurocomputing*, *308*, 49-57.

[54] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735-1780.

[55] Lee, J. Y., & Dernoncourt, F. (2016). Sequential short-text classification with recurrent and convolutional neural networks. *arXiv preprint arXiv:1603.03827*.

[56] Conneau, A., Schwenk, H., Barrault, L., & Lecun, Y. (2016). Very deep convolutional networks for text classification. *arXiv preprint arXiv:1606.01781*.

[57] Zhou, C., Sun, C., Liu, Z., & Lau, F. (2015). A C-LSTM neural network for text classification. *arXiv preprint arXiv:1511.08630*.

[58] Zhang, J., Li, Y., Tian, J., & Li, T. (2018, October). LSTM-CNN Hybrid Model for Text Classification. In *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)* (pp. 1675-1680). IEEE.

[59] Yin, W., Kann, K., Yu, M., & Schütze, H. (2017). Comparative study of CNN and RNN for natural language processing. *arXiv preprint arXiv:1702.01923*.

[60] LeCun, Y. (2015). LeNet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, *20*(5), 14.

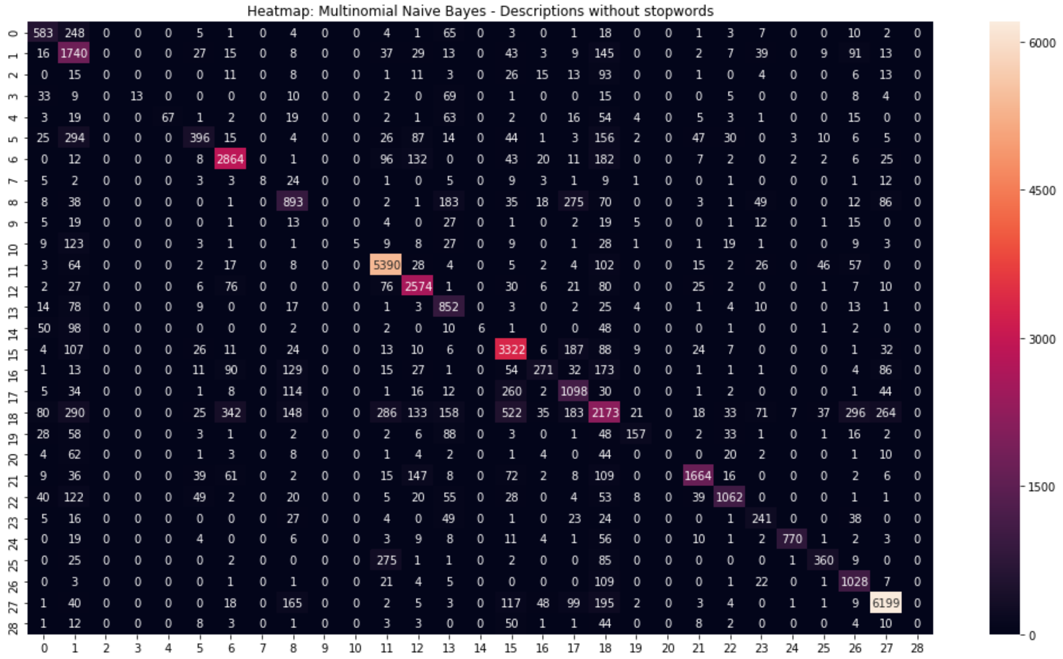
[61] UK government’s Find A Job Homepage, available at: https://www.gov.uk/find-a-job, last accessed January 2021

[62] GloVe Homepage, Available at https://nlp.stanford.edu/projects/glove/, last accessed January 2021

[63] Keras Official Website, available at: https://keras.io/, last accessed on January 2021

[64] Tensorflow Official Website, available at: https://www.tensorflow.org/, last accessed on January 2021

APPENDIX



Heatmap: Multinomial Naive Bayes - Descriptions without stopwords

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	583	248	0	0	0	5	1	0	4	0	0	4	1	65	0	3	0	1	18	0	0	1	3	7	0	0	10	2	0
1	16	1740	0	0	0	27	15	0	8	0	0	37	29	13	0	43	3	9	145	0	0	2	7	39	0	9	91	13	0
2	0	15	0	0	0	0	11	0	8	0	0	1	11	3	0	26	15	13	93	0	0	1	0	4	0	0	6	13	0
3	33	9	0	13	0	0	0	0	10	0	0	2	0	69	0	1	0	0	15	0	0	0	5	0	0	0	8	4	0
4	3	19	0	0	67	1	2	0	19	0	0	2	1	63	0	2	0	16	54	4	0	5	3	1	0	0	15	0	0
5	25	294	0	0	0	396	15	0	4	0	0	26	87	14	0	44	1	3	156	2	0	47	30	0	3	10	6	5	0
6	0	12	0	0	0	8	2864	0	1	0	0	96	132	0	0	43	20	11	182	0	0	7	2	0	2	2	6	25	0
7	5	2	0	0	0	3	3	8	24	0	0	1	0	5	0	9	3	1	9	1	0	0	1	0	0	0	1	12	0
8	8	38	0	0	0	0	1	0	893	0	0	2	1	183	0	35	18	275	70	0	0	3	1	49	0	0	12	86	0
9	5	19	0	0	0	0	1	0	13	0	0	4	0	27	0	1	0	2	19	5	0	0	1	12	0	1	15	0	0
10	9	123	0	0	0	3	1	0	1	0	5	9	8	27	0	9	0	1	28	1	0	1	19	1	0	0	9	3	0
11	3	64	0	0	0	2	17	0	8	0	0	5390	28	4	0	5	2	4	102	0	0	15	2	26	0	46	57	0	0
12	2	27	0	0	0	6	76	0	0	0	0	76	2574	1	0	30	6	21	80	0	0	25	2	0	0	1	7	10	0
13	14	78	0	0	0	9	0	0	17	0	0	1	3	852	0	3	0	2	25	4	0	1	4	10	0	0	13	1	0
14	50	98	0	0	0	0	0	0	2	0	0	2	0	10	6	1	0	0	48	0	0	0	1	0	0	1	2	0	0
15	4	107	0	0	0	26	11	0	24	0	0	13	10	6	0	3322	6	187	88	9	0	24	7	0	0	0	1	32	0
16	1	13	0	0	0	11	90	0	129	0	0	15	27	1	0	54	271	32	173	0	0	1	1	1	0	0	4	86	0
17	5	34	0	0	0	1	8	0	114	0	0	1	16	12	0	260	2	1098	30	0	0	1	2	0	0	0	1	44	0
18	80	290	0	0	0	25	342	0	148	0	0	286	133	158	0	522	35	183	2173	21	0	18	33	71	7	37	296	264	0
19	28	58	0	0	0	3	1	0	2	0	0	2	6	88	0	3	0	1	48	157	0	2	33	1	0	1	16	2	0
20	4	62	0	0	0	1	3	0	8	0	0	1	4	2	0	1	4	0	44	0	0	0	20	2	0	0	1	10	0
21	9	36	0	0	0	39	61	0	2	0	0	15	147	8	0	72	2	8	109	0	0	1664	16	0	0	0	2	6	0
22	40	122	0	0	0	49	2	0	20	0	0	5	20	55	0	28	0	4	53	8	0	39	1062	0	0	0	1	1	0
23	5	16	0	0	0	0	0	0	27	0	0	4	0	49	0	1	0	23	24	0	0	0	1	241	0	0	38	0	0
24	0	19	0	0	0	4	0	0	6	0	0	3	9	8	0	11	4	1	56	0	0	10	1	2	770	1	2	3	0
25	0	25	0	0	0	0	2	0	0	0	0	275	1	1	0	2	0	0	85	0	0	0	0	0	1	360	9	0	0
26	0	3	0	0	0	0	1	0	1	0	0	21	4	5	0	0	0	0	109	0	0	0	1	22	0	1	1028	7	0
27	1	40	0	0	0	0	18	0	165	0	0	2	5	3	0	117	48	99	195	2	0	3	4	0	1	1	9	6199	0
28	1	12	0	0	0	8	3	0	1	0	0	3	3	0	0	50	1	1	44	0	0	8	2	0	0	0	4	10	0

Figure 60: Confusion Matrix of MNB on BoW and lower case descriptions

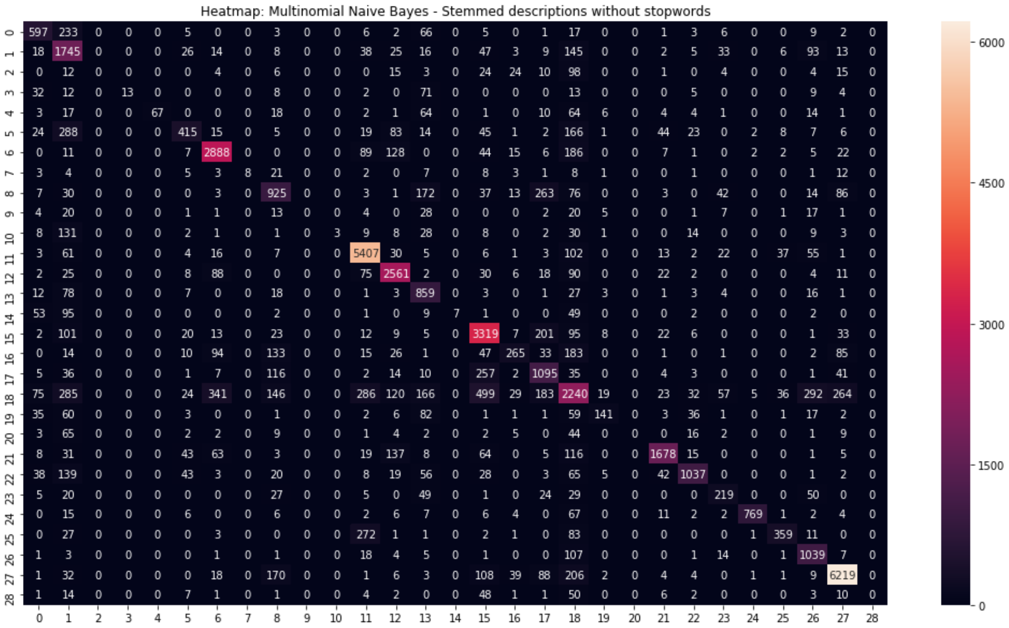


Figure 61: Confusion Matrix of MNB on BoW and lower case and stemmed descriptions

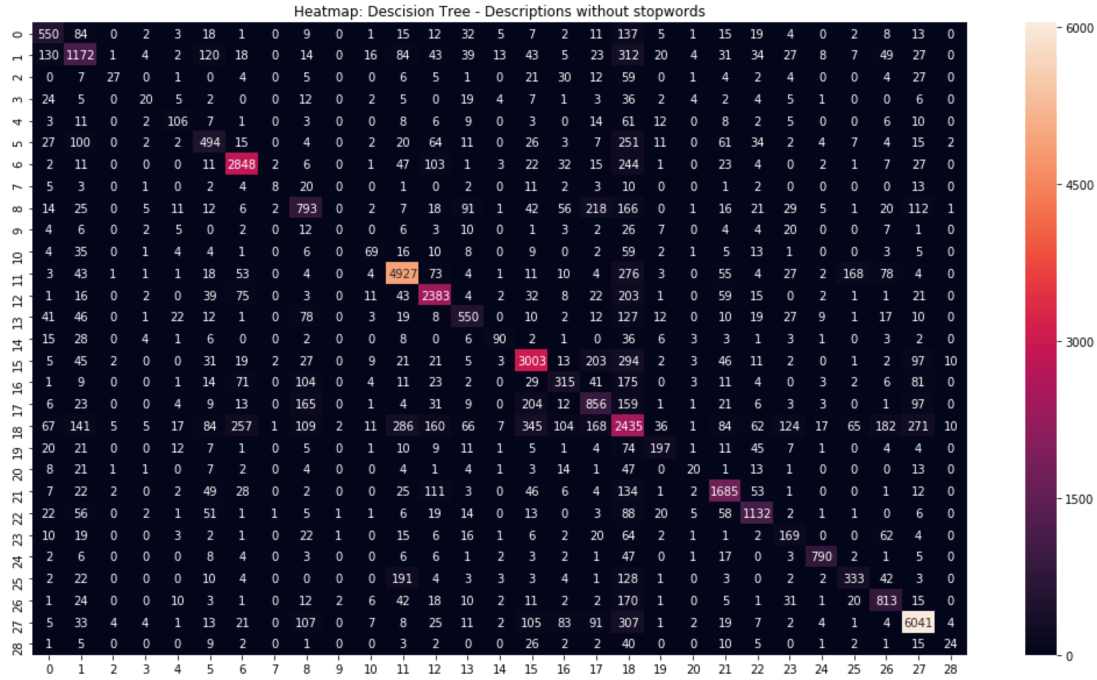
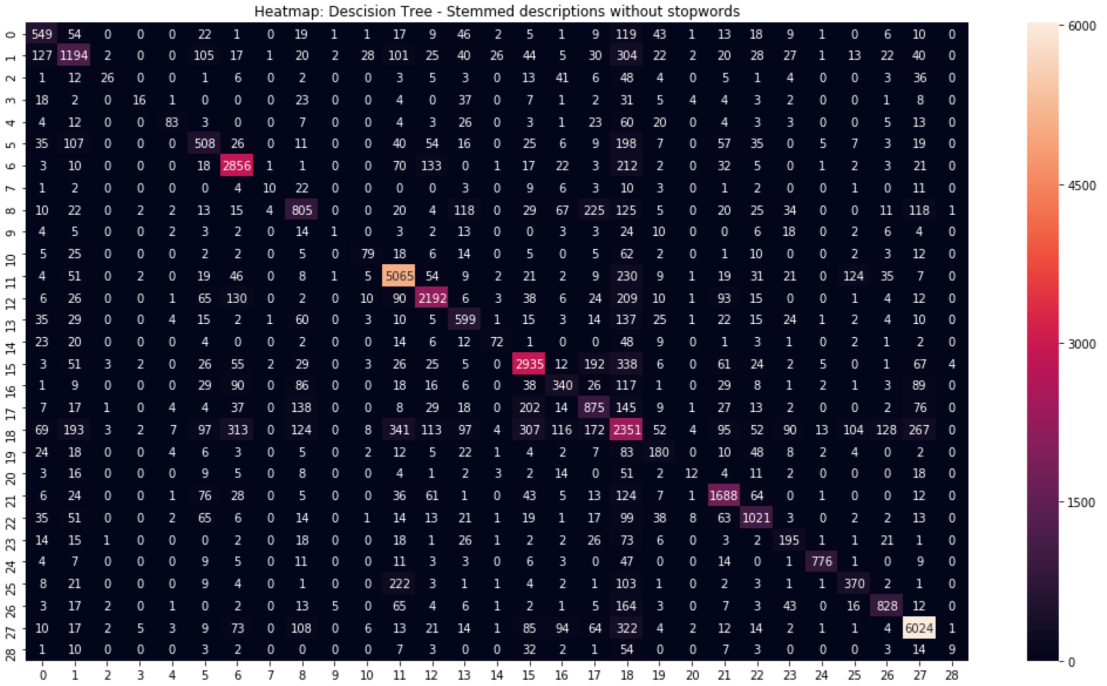
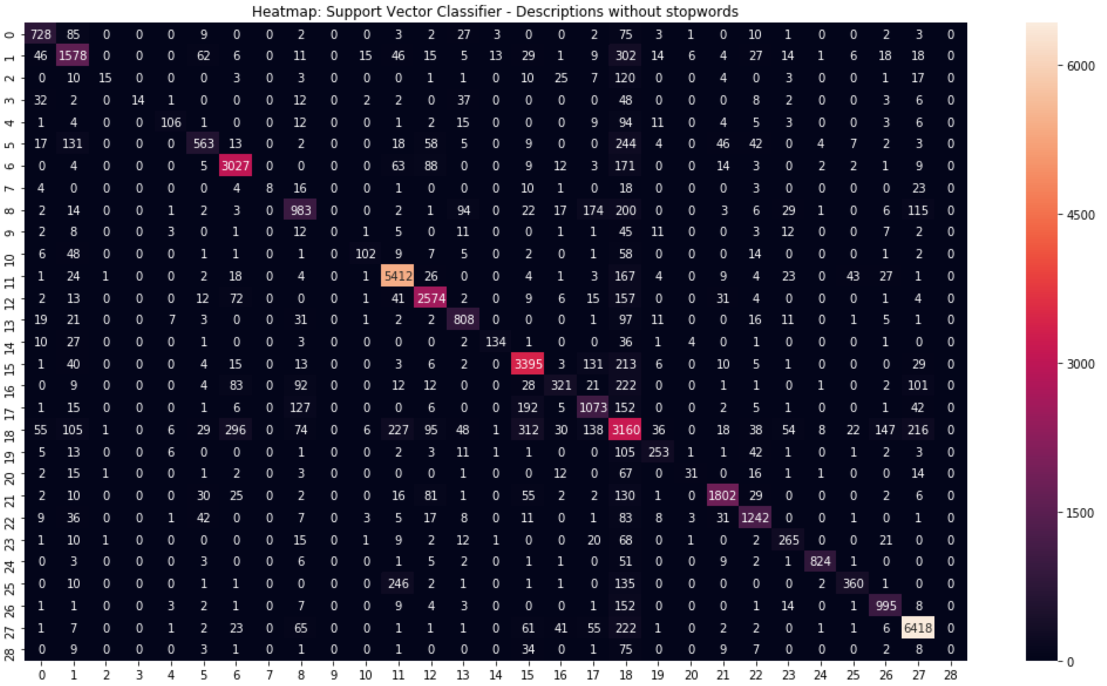


Figure 62: Confusion Matrix of DT on BoW and lower case descriptions



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	549	54	0	0	0	22	1	0	19	1	1	17	9	46	2	5	1	9	119	43	1	13	18	9	1	0	6	10	0
1	127	1194	2	0	0	105	17	1	20	2	28	101	25	40	26	44	5	30	304	22	2	20	28	27	1	13	22	40	0
2	1	12	26	0	0	1	6	0	2	0	0	3	5	3	0	13	41	6	48	4	0	5	1	4	0	0	3	36	0
3	18	2	0	16	1	0	0	0	23	0	0	4	0	37	0	7	1	2	31	5	4	4	3	2	0	0	1	8	0
4	4	12	0	0	83	3	0	0	7	0	0	4	3	26	0	3	1	23	60	20	0	4	3	3	0	0	5	13	0
5	35	107	0	0	0	508	26	0	11	0	0	40	54	16	0	25	6	9	198	7	0	57	35	0	5	7	3	19	0
6	3	10	0	0	0	18	2856	1	1	0	0	70	133	0	1	17	22	3	212	2	0	32	5	0	1	2	3	21	0
7	1	2	0	0	0	0	4	10	22	0	0	0	0	3	0	9	6	3	10	3	0	1	2	0	0	1	0	11	0
8	10	22	0	2	2	13	15	4	805	0	0	20	4	118	0	29	67	225	125	5	0	20	25	34	0	0	11	118	1
9	4	5	0	0	2	3	2	0	14	1	0	3	2	13	0	0	3	3	24	10	0	0	6	18	0	2	6	4	0
10	5	25	0	0	0	2	2	0	5	0	79	18	6	14	0	5	0	5	62	2	0	1	10	0	0	2	3	12	0
11	4	51	0	2	0	19	46	0	8	1	5	5065	54	9	2	21	2	9	230	9	1	19	31	21	0	124	35	7	0
12	6	26	0	0	1	65	130	0	2	0	10	90	2192	6	3	38	6	24	209	10	1	93	15	0	0	1	4	12	0
13	35	29	0	0	4	15	2	1	60	0	3	10	5	599	1	15	3	14	137	25	1	22	15	24	1	2	4	10	0
14	23	20	0	0	0	4	0	0	2	0	0	14	6	12	72	1	0	0	48	9	0	1	3	1	0	2	1	2	0
15	3	51	3	2	0	26	55	2	29	0	3	26	25	5	0	2935	12	192	338	6	0	61	24	2	5	0	1	67	4
16	1	9	0	0	0	29	90	0	86	0	0	18	16	6	0	38	340	26	117	1	0	29	8	1	2	1	3	89	0
17	7	17	1	0	4	4	37	0	138	0	0	8	29	18	0	202	14	875	145	9	1	27	13	2	0	0	2	76	0
18	69	193	3	2	7	97	313	0	124	0	8	341	113	97	4	307	116	172	2351	52	4	95	52	90	13	104	128	267	0
19	24	18	0	0	4	6	3	0	5	0	2	12	5	22	1	4	2	7	83	180	0	10	48	8	2	4	0	2	0
20	3	16	0	0	0	9	5	0	8	0	0	4	1	2	3	2	14	0	51	2	12	4	11	2	0	0	0	18	0
21	6	24	0	0	1	76	28	0	5	0	0	36	61	1	0	43	5	13	124	7	1	1688	64	0	1	0	0	12	0
22	35	51	0	0	2	65	6	0	14	0	1	14	13	21	1	19	1	17	99	38	8	63	1021	3	0	2	2	13	0
23	14	15	1	0	0	0	2	0	18	0	0	18	1	26	1	2	2	26	73	6	0	3	2	195	1	1	21	1	0
24	4	7	0	0	0	9	5	0	11	0	0	11	3	3	0	6	3	0	47	0	0	14	0	1	776	1	0	9	0
25	8	21	0	0	0	9	4	0	1	0	0	222	3	1	1	4	2	1	103	1	0	2	3	1	1	370	2	1	0
26	3	17	2	0	1	0	2	0	13	5	0	65	4	6	1	2	1	5	164	3	0	7	3	43	0	16	828	12	0
27	10	17	2	5	3	9	73	0	108	0	6	13	21	14	1	85	94	64	322	4	2	12	14	2	1	1	4	6024	1
28	1	10	0	0	0	3	2	0	0	0	0	7	3	0	0	32	2	1	54	0	0	7	3	0	0	0	3	14	9

Figure 63: Confusion Matrix of DT on BoW and lower case and stemmed descriptions



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	728	85	0	0	0	9	0	0	2	0	0	3	2	27	3	0	0	2	75	3	1	0	10	1	0	0	2	3	0
1	46	1578	0	0	0	62	6	0	11	0	15	46	15	5	13	29	1	9	302	14	6	4	27	14	1	6	18	18	0
2	0	10	15	0	0	0	3	0	3	0	0	0	1	1	0	10	25	7	120	0	0	4	0	3	0	0	1	17	0
3	32	2	0	14	1	0	0	0	12	0	2	2	0	37	0	0	0	0	48	0	0	0	8	2	0	0	3	6	0
4	1	4	0	0	106	1	0	0	12	0	0	1	2	15	0	0	0	9	94	11	0	4	5	3	0	0	3	6	0
5	17	131	0	0	0	563	13	0	2	0	0	18	58	5	0	9	0	0	244	4	0	46	42	0	4	7	2	3	0
6	0	4	0	0	0	5	3027	0	0	0	0	63	88	0	0	9	12	3	171	0	0	14	3	0	2	2	1	9	0
7	4	0	0	0	0	0	4	8	16	0	0	1	0	0	0	10	1	0	18	0	0	0	3	0	0	0	0	23	0
8	2	14	0	0	1	2	3	0	983	0	0	2	1	94	0	22	17	174	200	0	0	3	6	29	1	0	6	115	0
9	2	8	0	0	3	0	1	0	12	0	1	5	0	11	0	0	1	1	45	11	0	0	3	12	0	0	7	2	0
10	6	48	0	0	0	1	1	0	1	0	102	9	7	5	0	2	0	1	58	0	0	0	14	0	0	0	1	2	0
11	1	24	1	0	0	2	18	0	4	0	1	5412	26	0	0	4	1	3	167	4	0	9	4	23	0	43	27	1	0
12	2	13	0	0	0	12	72	0	0	0	1	41	2574	2	0	9	6	15	157	0	0	31	4	0	0	0	1	4	0
13	19	21	0	0	7	3	0	0	31	0	1	2	2	808	0	0	0	1	97	11	0	0	16	11	0	1	5	1	0
14	10	27	0	0	0	1	0	0	3	0	0	0	0	2	134	1	0	0	36	1	4	0	1	0	0	0	1	0	0
15	1	40	0	0	0	4	15	0	13	0	0	3	6	2	0	3395	3	131	213	6	0	10	5	1	0	0	0	29	0
16	0	9	0	0	0	4	83	0	92	0	0	12	12	0	0	28	321	21	222	0	0	1	1	0	1	0	2	101	0
17	1	15	0	0	0	1	6	0	127	0	0	0	6	0	0	192	5	1073	152	0	0	2	5	1	0	0	1	42	0
18	55	105	1	0	6	29	296	0	74	0	6	227	95	48	1	312	30	138	3160	36	0	18	38	54	8	22	147	216	0
19	5	13	0	0	6	0	0	0	1	0	0	2	3	11	1	1	0	0	105	253	1	1	42	1	0	1	2	3	0
20	2	15	1	0	0	1	2	0	3	0	0	0	0	1	0	0	12	0	67	0	31	0	16	1	1	0	0	14	0
21	2	10	0	0	0	30	25	0	2	0	0	16	81	1	0	55	2	2	130	1	0	1802	29	0	0	0	2	6	0
22	9	36	0	0	1	42	0	0	7	0	3	5	17	8	0	11	0	1	83	8	3	31	1242	0	0	1	0	1	0
23	1	10	1	0	0	0	0	0	15	0	1	9	2	12	1	0	0	20	68	0	1	0	2	265	0	0	21	0	0
24	0	3	0	0	0	3	0	0	6	0	0	1	5	2	0	1	1	0	51	0	0	9	2	1	824	1	0	0	0
25	0	10	0	0	0	1	1	0	0	0	0	246	2	1	0	1	1	0	135	0	0	0	0	0	2	360	1	0	0
26	1	1	0	0	3	2	1	0	7	0	0	9	4	3	0	0	0	1	152	0	0	0	1	14	0	1	995	8	0
27	1	7	0	0	1	2	23	0	65	0	0	1	1	1	0	61	41	55	222	1	0	2	2	0	1	1	6	6418	0
28	0	9	0	0	0	3	1	0	1	0	0	1	0	0	0	34	0	1	75	0	0	9	7	0	0	0	2	8	0

Figure 64: Confusion Matrix of SVC on BoW and lower case descriptions

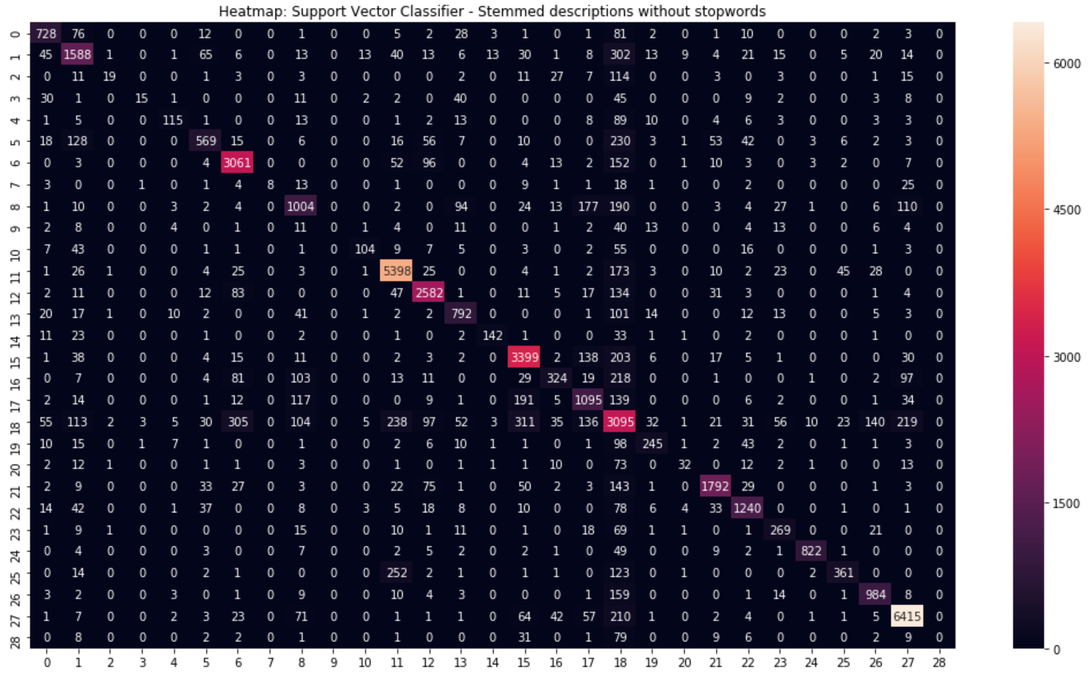


Figure 65: Confusion Matrix of SVC on BoW and lower case and stemmed descriptions

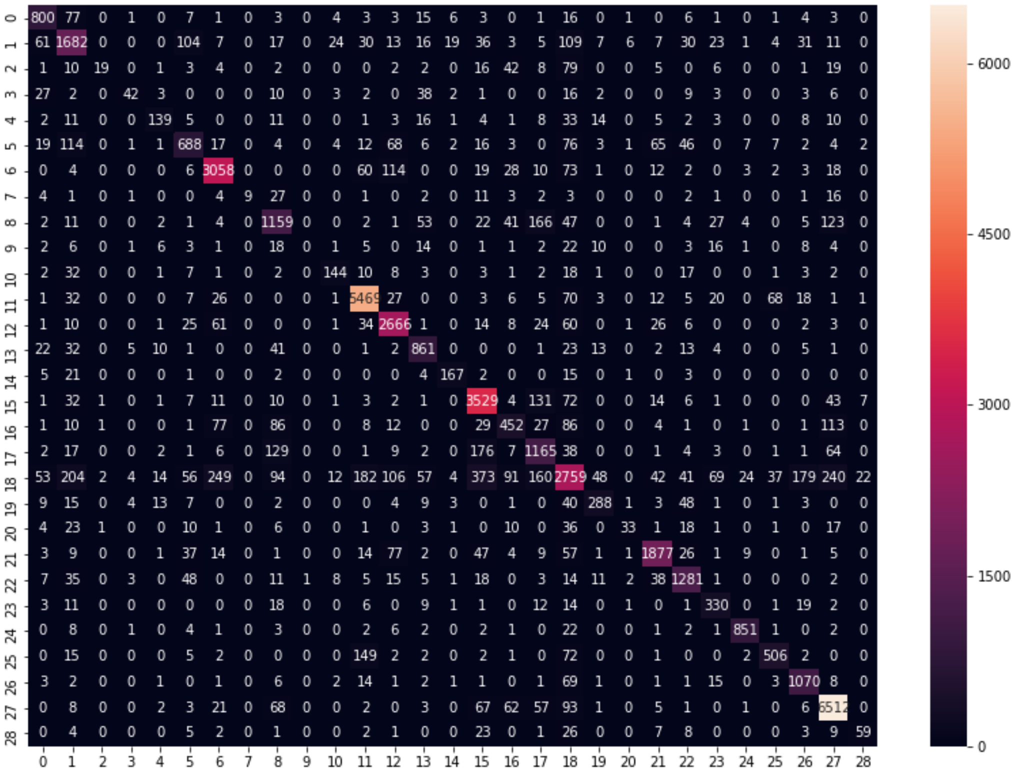
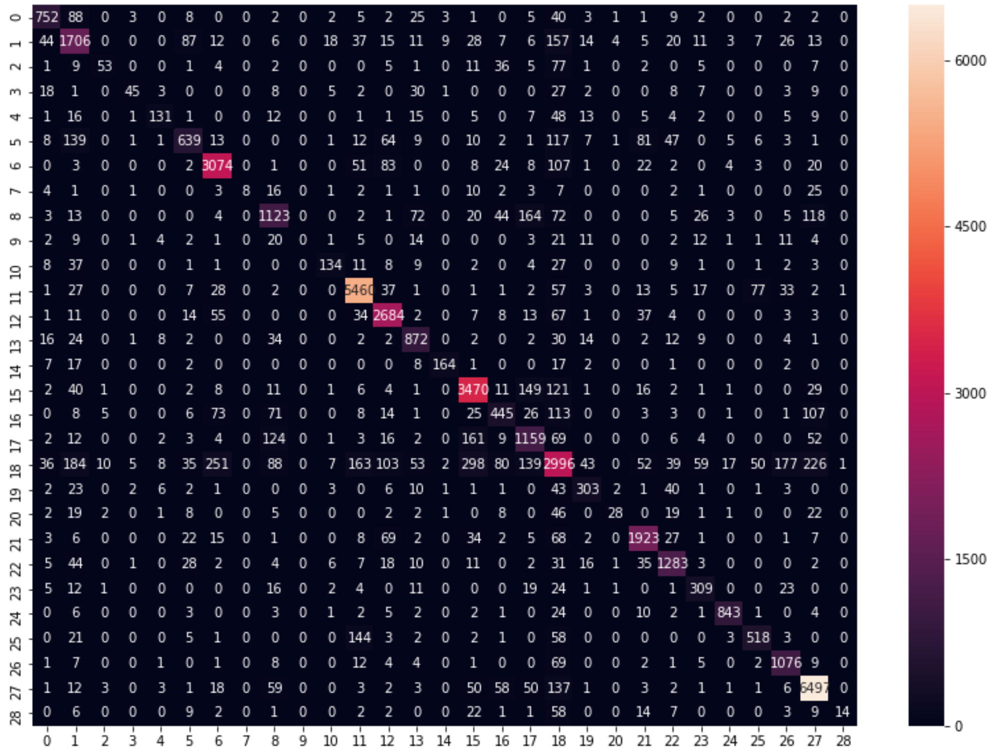
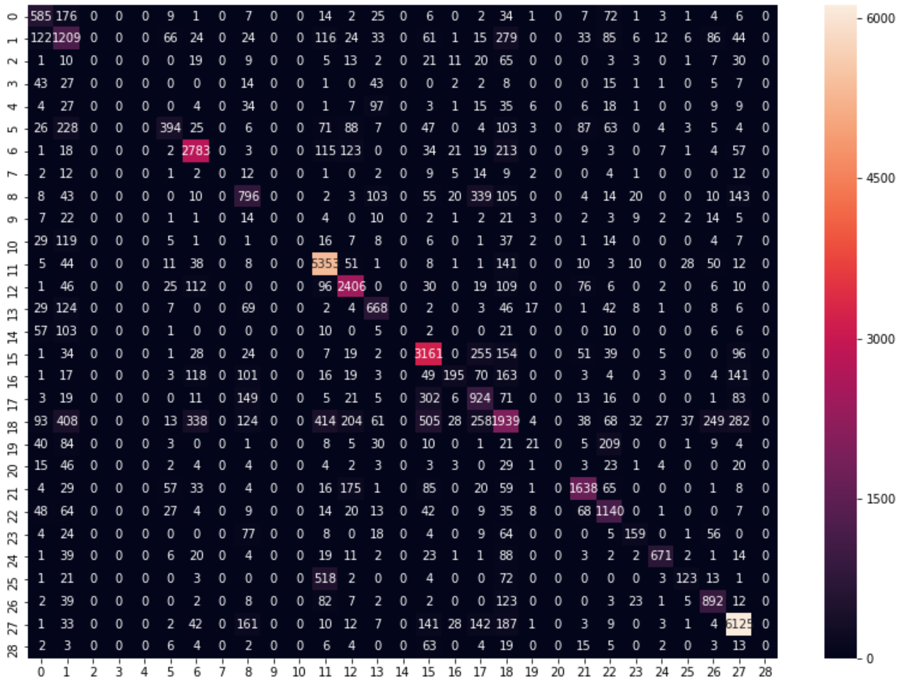


Figure 66: Confusion Matrix of FNN on BoW and lower case descriptions



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	752	88	0	3	0	8	0	0	2	0	2	5	2	25	3	1	0	5	40	3	1	1	9	2	0	0	2	2	0
1	44	1706	0	0	0	87	12	0	6	0	18	37	15	11	9	28	7	6	157	14	4	5	20	11	3	7	26	13	0
2	1	9	53	0	0	1	4	0	2	0	0	0	5	1	0	11	36	5	77	1	0	2	0	5	0	0	0	7	0
3	18	1	0	45	3	0	0	0	8	0	5	2	0	30	1	0	0	0	27	2	0	0	8	7	0	0	3	9	0
4	1	16	0	1	131	1	0	0	12	0	0	1	1	15	0	5	0	7	48	13	0	5	4	2	0	0	5	9	0
5	8	139	0	1	1	639	13	0	0	0	1	12	64	9	0	10	2	1	117	7	1	81	47	0	5	6	3	1	0
6	0	3	0	0	0	2	3074	0	1	0	0	51	83	0	0	8	24	8	107	1	0	22	2	0	4	3	0	20	0
7	4	1	0	1	0	0	3	8	16	0	1	2	1	1	0	10	2	3	7	0	0	0	2	1	0	0	0	25	0
8	3	13	0	0	0	0	4	0	1123	0	0	2	1	72	0	20	44	164	72	0	0	0	5	26	3	0	5	118	0
9	2	9	0	1	4	2	1	0	20	0	1	5	0	14	0	0	0	3	21	11	0	0	2	12	1	1	11	4	0
10	8	37	0	0	0	1	1	0	0	0	134	11	8	9	0	2	0	4	27	0	0	0	9	1	0	1	2	3	0
11	1	27	0	0	0	7	28	0	2	0	0	5460	37	1	0	1	1	2	57	3	0	13	5	17	0	77	33	2	1
12	1	11	0	0	0	14	55	0	0	0	0	34	2684	2	0	7	8	13	67	1	0	37	4	0	0	0	3	3	0
13	16	24	0	1	8	2	0	0	34	0	0	2	2	872	0	2	0	2	30	14	0	2	12	9	0	0	4	1	0
14	7	17	0	0	0	2	0	0	0	0	0	0	0	8	164	1	0	0	17	2	0	0	1	0	0	0	2	0	0
15	2	40	1	0	0	2	8	0	11	0	1	6	4	1	0	3470	11	149	121	1	0	16	2	1	1	0	0	29	0
16	0	8	5	0	0	6	73	0	71	0	0	8	14	1	0	25	445	26	113	0	0	3	3	0	1	0	1	107	0
17	2	12	0	0	2	3	4	0	124	0	1	3	16	2	0	161	9	1159	69	0	0	0	6	4	0	0	0	52	0
18	36	184	10	5	8	35	251	0	88	0	7	163	103	53	2	298	80	139	2996	43	0	52	39	59	17	50	177	226	1
19	2	23	0	2	6	2	1	0	0	0	3	0	6	10	1	1	1	0	43	303	2	1	40	1	0	1	3	0	0
20	2	19	2	0	1	8	0	0	5	0	0	0	2	2	1	0	8	0	46	0	28	0	19	1	1	0	0	22	0
21	3	6	0	0	0	22	15	0	1	0	0	8	69	2	0	34	2	5	68	2	0	1923	27	1	0	0	1	7	0
22	5	44	0	1	0	28	2	0	4	0	6	7	18	10	0	11	0	2	31	16	1	35	1283	3	0	0	0	2	0
23	5	12	1	0	0	0	0	0	16	0	2	4	0	11	0	0	0	19	24	1	1	0	1	309	0	0	23	0	0
24	0	6	0	0	0	3	0	0	3	0	1	2	5	2	0	2	1	0	24	0	0	10	2	1	843	1	0	4	0
25	0	21	0	0	0	5	1	0	0	0	0	144	3	2	0	2	1	0	58	0	0	0	0	0	3	518	3	0	0
26	1	7	0	0	1	0	1	0	8	0	0	12	4	4	0	1	0	0	69	0	0	2	1	5	0	2	1076	9	0
27	1	12	3	0	3	1	18	0	59	0	0	3	2	3	0	50	58	50	137	1	0	3	2	1	1	1	6	6497	0
28	0	6	0	0	0	9	2	0	1	0	0	2	2	0	0	22	1	1	58	0	0	14	7	0	0	0	3	9	14

Figure 67: Confusion Matrix of FNN on BoW and lower case and stemmed descriptions



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	585	176	0	0	0	9	1	0	7	0	0	14	2	25	0	6	0	2	34	1	0	7	72	1	3	1	4	6	0
1	122	1209	0	0	0	66	24	0	24	0	0	116	24	33	0	61	1	15	279	0	0	33	85	6	12	6	86	44	0
2	1	10	0	0	0	0	19	0	9	0	0	5	13	2	0	21	11	20	65	0	0	0	3	3	0	1	7	30	0
3	43	27	0	0	0	0	0	0	14	0	0	1	0	43	0	0	2	2	8	0	0	0	15	1	1	0	5	7	0
4	4	27	0	0	0	0	4	0	34	0	0	1	7	97	0	3	1	15	35	6	0	6	18	1	0	0	9	9	0
5	26	228	0	0	0	394	25	0	6	0	0	71	88	7	0	47	0	4	103	3	0	87	63	0	4	3	5	4	0
6	1	18	0	0	0	2	2783	0	3	0	0	115	123	0	0	34	21	19	213	0	0	9	3	0	7	1	4	57	0
7	2	12	0	0	0	1	2	0	12	0	0	1	0	2	0	9	5	14	9	2	0	0	4	1	0	0	0	12	0
8	8	43	0	0	0	0	10	0	796	0	0	2	3	103	0	55	20	339	105	0	0	4	14	20	0	0	10	143	0
9	7	22	0	0	0	1	1	0	14	0	0	4	0	10	0	2	1	2	21	3	0	2	3	9	2	2	14	5	0
10	29	119	0	0	0	5	1	0	1	0	0	16	7	8	0	6	0	1	37	2	0	1	14	0	0	0	4	7	0
11	5	44	0	0	0	11	38	0	8	0	0	5353	51	1	0	8	1	1	141	0	0	10	3	10	0	28	50	12	0
12	1	46	0	0	0	25	112	0	0	0	0	96	2406	0	0	30	0	19	109	0	0	76	6	0	2	0	6	10	0
13	29	124	0	0	0	7	0	0	69	0	0	2	4	668	0	2	0	3	46	17	0	1	42	8	1	0	8	6	0
14	57	103	0	0	0	1	0	0	0	0	0	10	0	5	0	2	0	0	21	0	0	0	10	0	0	0	6	6	0
15	1	34	0	0	0	1	28	0	24	0	0	7	19	2	0	3161	0	255	154	0	0	51	39	0	5	0	0	96	0
16	1	17	0	0	0	3	118	0	101	0	0	16	19	3	0	49	195	70	163	0	0	3	4	0	3	0	4	141	0
17	3	19	0	0	0	0	11	0	149	0	0	5	21	5	0	302	6	924	71	0	0	13	16	0	0	0	1	83	0
18	93	408	0	0	0	13	338	0	124	0	0	414	204	61	0	505	28	258	1939	4	0	38	68	32	27	37	249	282	0
19	40	84	0	0	0	3	0	0	1	0	0	8	5	30	0	10	0	1	21	21	0	5	209	0	0	1	9	4	0
20	15	46	0	0	0	2	4	0	4	0	0	4	2	3	0	3	3	0	29	1	0	3	23	1	4	0	0	20	0
21	4	29	0	0	0	57	33	0	4	0	0	16	175	1	0	85	0	20	59	1	0	1638	65	0	0	0	1	8	0
22	48	64	0	0	0	27	4	0	9	0	0	14	20	13	0	42	0	9	35	8	0	68	1140	0	1	0	0	7	0
23	4	24	0	0	0	0	0	0	77	0	0	8	0	18	0	4	0	9	64	0	0	0	5	159	0	1	56	0	0
24	1	39	0	0	0	6	20	0	4	0	0	19	11	2	0	23	1	1	88	0	0	3	2	2	671	2	1	14	0
25	1	21	0	0	0	0	3	0	0	0	0	518	2	0	0	4	0	0	72	0	0	0	0	0	3	123	13	1	0
26	2	39	0	0	0	0	2	0	8	0	0	82	7	2	0	2	0	0	123	0	0	0	3	23	1	5	892	12	0
27	1	33	0	0	0	2	42	0	161	0	0	10	12	7	0	141	28	142	187	1	0	3	9	0	3	1	4	6125	0
28	2	3	0	0	0	6	4	0	2	0	0	6	4	0	0	63	0	4	19	0	0	15	5	0	2	0	3	13	0

Figure 68: Confusion Matrix of FNN trained on Google's word2vec (Glove) embeddings with lower case descriptions

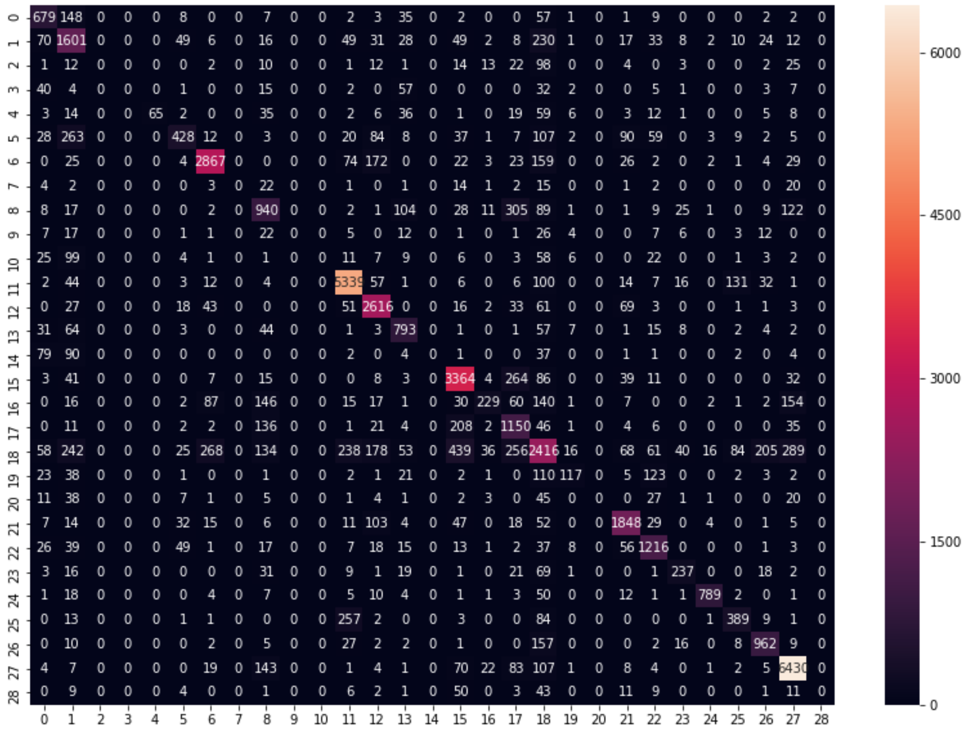


Figure 69: Confusion Matrix of FNN trained on custom word2vec embeddings (with skipgram) with lower case descriptions

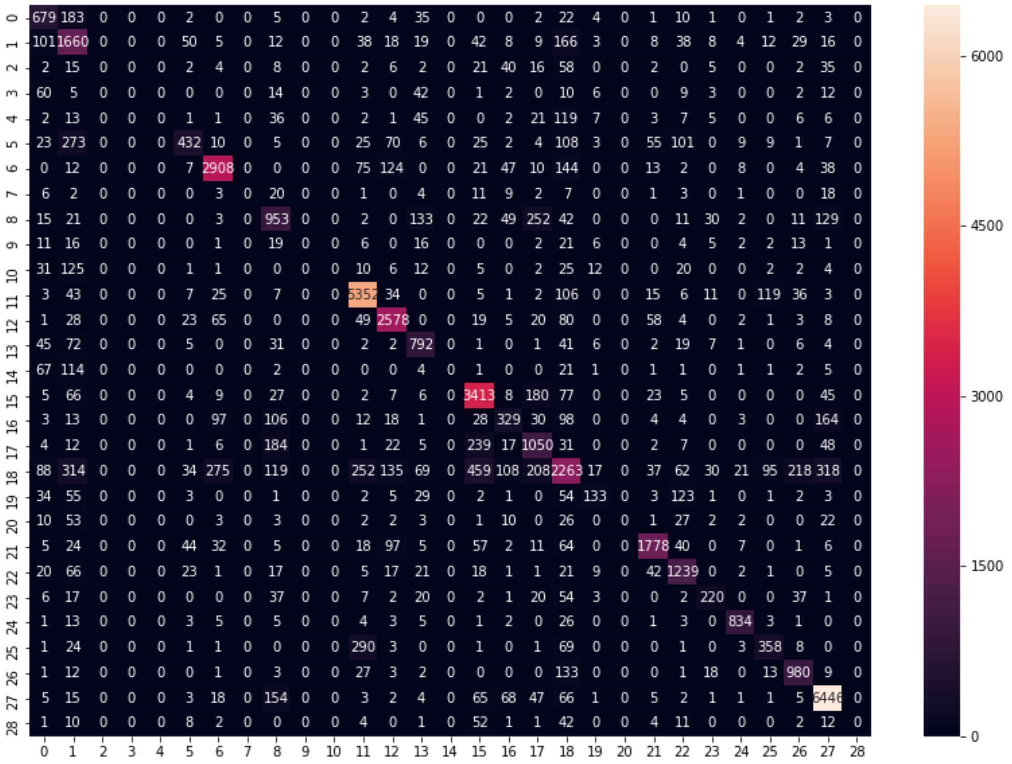
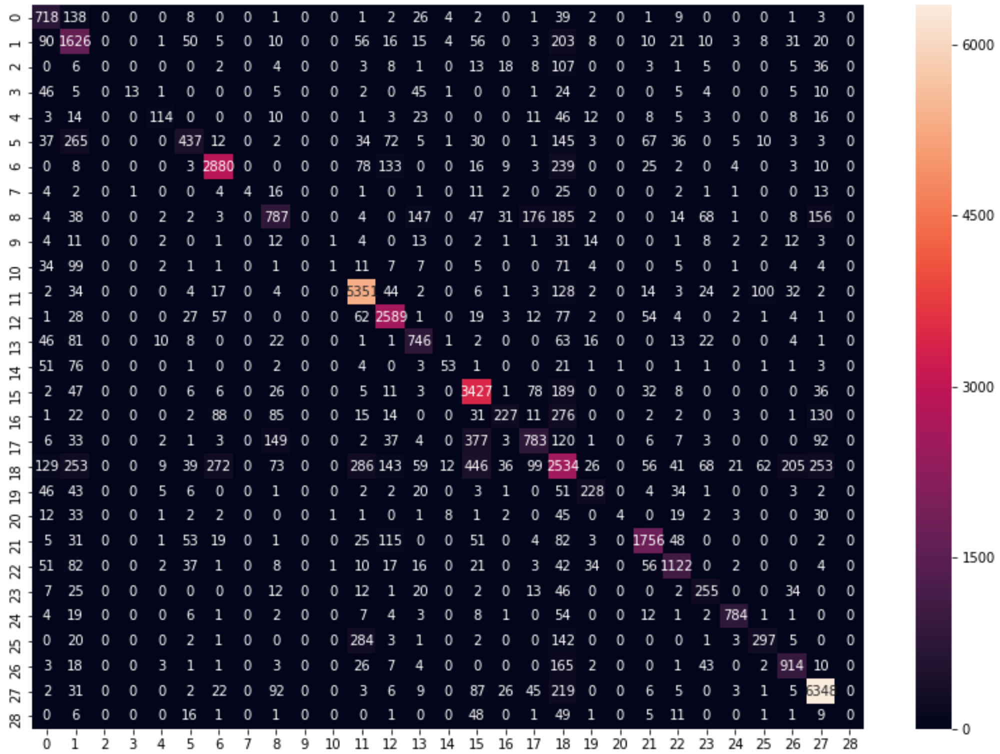
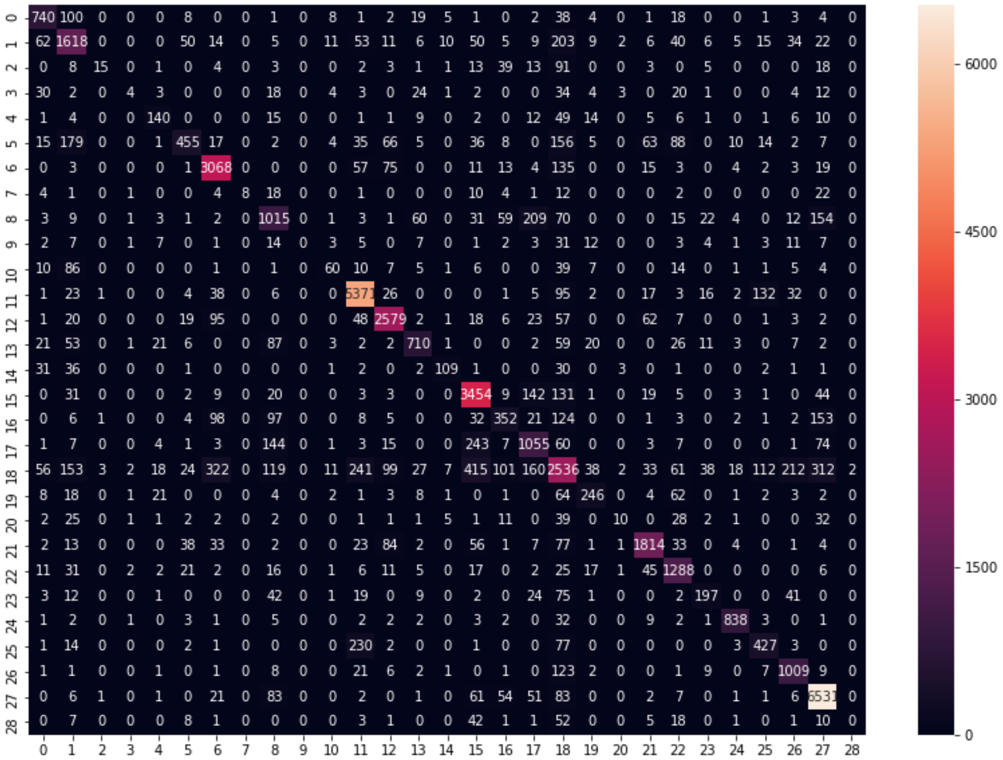


Figure 70: Confusion Matrix of FNN trained on custom FastText embeddings (with skipgram) with lower case descriptions



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	718	138	0	0	0	8	0	0	1	0	0	1	2	26	4	2	0	1	39	2	0	1	9	0	0	0	1	3	0
1	90	1626	0	0	1	50	5	0	10	0	0	56	16	15	4	56	0	3	203	8	0	10	21	10	3	8	31	20	0
2	0	6	0	0	0	0	2	0	4	0	0	3	8	1	0	13	18	8	107	0	0	3	1	5	0	0	5	36	0
3	46	5	0	13	1	0	0	0	5	0	0	2	0	45	1	0	0	1	24	2	0	0	5	4	0	0	5	10	0
4	3	14	0	0	114	0	0	0	10	0	0	1	3	23	0	0	0	11	46	12	0	8	5	3	0	0	8	16	0
5	37	265	0	0	0	437	12	0	2	0	0	34	72	5	1	30	0	1	145	3	0	67	36	0	5	10	3	3	0
6	0	8	0	0	0	3	2880	0	0	0	0	78	133	0	0	16	9	3	239	0	0	25	2	0	4	0	3	10	0
7	4	2	0	1	0	0	4	4	16	0	0	1	0	1	0	11	2	0	25	0	0	0	2	1	1	0	0	13	0
8	4	38	0	0	2	2	3	0	787	0	0	4	0	147	0	47	31	176	185	2	0	0	14	68	1	0	8	156	0
9	4	11	0	0	2	0	1	0	12	0	1	4	0	13	0	2	1	1	31	14	0	0	1	8	2	2	12	3	0
10	34	99	0	0	2	1	1	0	1	0	1	11	7	7	0	5	0	0	71	4	0	0	5	0	1	0	4	4	0
11	2	34	0	0	0	4	17	0	4	0	0	5351	44	2	0	6	1	3	128	2	0	14	3	24	2	100	32	2	0
12	1	28	0	0	0	27	57	0	0	0	0	62	2589	1	0	19	3	12	77	2	0	54	4	0	2	1	4	1	0
13	46	81	0	0	10	8	0	0	22	0	0	1	1	746	1	2	0	0	63	16	0	0	13	22	0	0	4	1	0
14	51	76	0	0	0	1	0	0	2	0	0	4	0	3	53	1	0	0	21	1	1	0	1	1	0	1	1	3	0
15	2	47	0	0	0	6	6	0	26	0	0	5	11	3	0	3427	1	78	189	0	0	32	8	0	0	0	0	36	0
16	1	22	0	0	0	2	88	0	85	0	0	15	14	0	0	31	227	11	276	0	0	2	2	0	3	0	1	130	0
17	6	33	0	0	2	1	3	0	149	0	0	2	37	4	0	377	3	783	120	1	0	6	7	3	0	0	0	92	0
18	129	253	0	0	9	39	272	0	73	0	0	286	143	59	12	446	36	99	2534	26	0	56	41	68	21	62	205	253	0
19	46	43	0	0	5	6	0	0	1	0	0	2	2	20	0	3	1	0	51	228	0	4	34	1	0	0	3	2	0
20	12	33	0	0	1	2	2	0	0	0	1	1	0	1	8	1	2	0	45	0	4	0	19	2	3	0	0	30	0
21	5	31	0	0	1	53	19	0	1	0	0	25	115	0	0	51	0	4	82	3	0	1756	48	0	0	0	0	2	0
22	51	82	0	0	2	37	1	0	8	0	1	10	17	16	0	21	0	3	42	34	0	56	1122	0	2	0	0	4	0
23	7	25	0	0	0	0	0	0	12	0	0	12	1	20	0	2	0	13	46	0	0	0	2	255	0	0	34	0	0
24	4	19	0	0	0	6	1	0	2	0	0	7	4	3	0	8	1	0	54	0	0	12	1	2	784	1	1	0	0
25	0	20	0	0	0	2	1	0	0	0	0	284	3	1	0	2	0	0	142	0	0	0	0	1	3	297	5	0	0
26	3	18	0	0	3	1	1	0	3	0	0	26	7	4	0	0	0	0	165	2	0	0	1	43	0	2	914	10	0
27	2	31	0	0	0	2	22	0	92	0	0	3	6	9	0	87	26	45	219	0	0	6	5	0	3	1	5	6348	0
28	0	6	0	0	0	16	1	0	1	0	0	0	1	0	0	48	0	1	49	1	0	5	11	0	0	1	1	9	0

Figure 71: Confusion Matrix of CNN trained on Google's word2vec (Glove) embeddings with lower case descriptions



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	740	100	0	0	0	8	0	0	1	0	8	1	2	19	5	1	0	2	38	4	0	1	18	0	0	1	3	4	0
1	62	1618	0	0	0	50	14	0	5	0	11	53	11	6	10	50	5	9	203	9	2	6	40	6	5	15	34	22	0
2	0	8	15	0	1	0	4	0	3	0	0	2	3	1	1	13	39	13	91	0	0	3	0	5	0	0	0	18	0
3	30	2	0	4	3	0	0	0	18	0	4	3	0	24	1	2	0	0	34	4	3	0	20	1	0	0	4	12	0
4	1	4	0	0	140	0	0	0	15	0	0	1	1	9	0	2	0	12	49	14	0	5	6	1	0	1	6	10	0
5	15	179	0	0	1	455	17	0	2	0	4	35	66	5	0	36	8	0	156	5	0	63	88	0	10	14	2	7	0
6	0	3	0	0	0	1	3068	0	0	0	0	57	75	0	0	11	13	4	135	0	0	15	3	0	4	2	3	19	0
7	4	1	0	1	0	0	4	8	18	0	0	1	0	0	0	10	4	1	12	0	0	0	2	0	0	0	0	22	0
8	3	9	0	1	3	1	2	0	1015	0	1	3	1	60	0	31	59	209	70	0	0	0	15	22	4	0	12	154	0
9	2	7	0	1	7	0	1	0	14	0	3	5	0	7	0	1	2	3	31	12	0	0	3	4	1	3	11	7	0
10	10	86	0	0	0	0	1	0	1	0	60	10	7	5	1	6	0	0	39	7	0	0	14	0	1	1	5	4	0
11	1	23	1	0	0	4	38	0	6	0	0	5371	26	0	0	0	1	5	95	2	0	17	3	16	2	132	32	0	0
12	1	20	0	0	0	19	95	0	0	0	0	48	2579	2	1	18	6	23	57	0	0	62	7	0	0	1	3	2	0
13	21	53	0	1	21	6	0	0	87	0	3	2	2	710	1	0	0	2	59	20	0	0	26	11	3	0	7	2	0
14	31	36	0	0	0	1	0	0	0	0	1	2	0	2	109	1	0	0	30	0	3	0	1	0	0	2	1	1	0
15	0	31	0	0	0	2	9	0	20	0	0	3	3	0	0	3454	9	142	131	1	0	19	5	0	3	1	0	44	0
16	0	6	1	0	0	4	98	0	97	0	0	8	5	0	0	32	352	21	124	0	0	1	3	0	2	1	2	153	0
17	1	7	0	0	4	1	3	0	144	0	1	3	15	0	0	243	7	1055	60	0	0	3	7	0	0	0	1	74	0
18	56	153	3	2	18	24	322	0	119	0	11	241	99	27	7	415	101	160	2536	38	2	33	61	38	18	112	212	312	2
19	8	18	0	1	21	0	0	0	4	0	2	1	3	8	1	0	1	0	64	246	0	4	62	0	1	2	3	2	0
20	2	25	0	1	1	2	2	0	2	0	0	1	1	1	5	1	11	0	39	0	10	0	28	2	1	0	0	32	0
21	2	13	0	0	0	38	33	0	2	0	0	23	84	2	0	56	1	7	77	1	1	1814	33	0	4	0	1	4	0
22	11	31	0	2	2	21	2	0	16	0	1	6	11	5	0	17	0	2	25	17	1	45	1288	0	0	0	0	6	0
23	3	12	0	0	1	0	0	0	42	0	1	19	0	9	0	2	0	24	75	1	0	0	2	197	0	0	41	0	0
24	1	2	0	1	0	3	1	0	5	0	0	2	2	2	0	3	2	0	32	0	0	9	2	1	838	3	0	1	0
25	1	14	0	0	0	2	1	0	0	0	0	230	2	0	0	1	0	0	77	0	0	0	0	0	3	427	3	0	0
26	1	1	0	0	1	0	1	0	8	0	0	21	6	2	1	0	1	0	123	2	0	0	1	9	0	7	1009	9	0
27	0	6	1	0	1	0	21	0	83	0	0	2	0	1	0	61	54	51	83	0	0	2	7	0	1	1	6	6531	0
28	0	7	0	0	0	8	1	0	0	0	0	3	1	0	0	42	1	1	52	0	0	5	18	0	1	0	1	10	0

Figure 72: Confusion Matrix of CNN trained on custom word2vec embeddings (with skipgram) with lower case descriptions

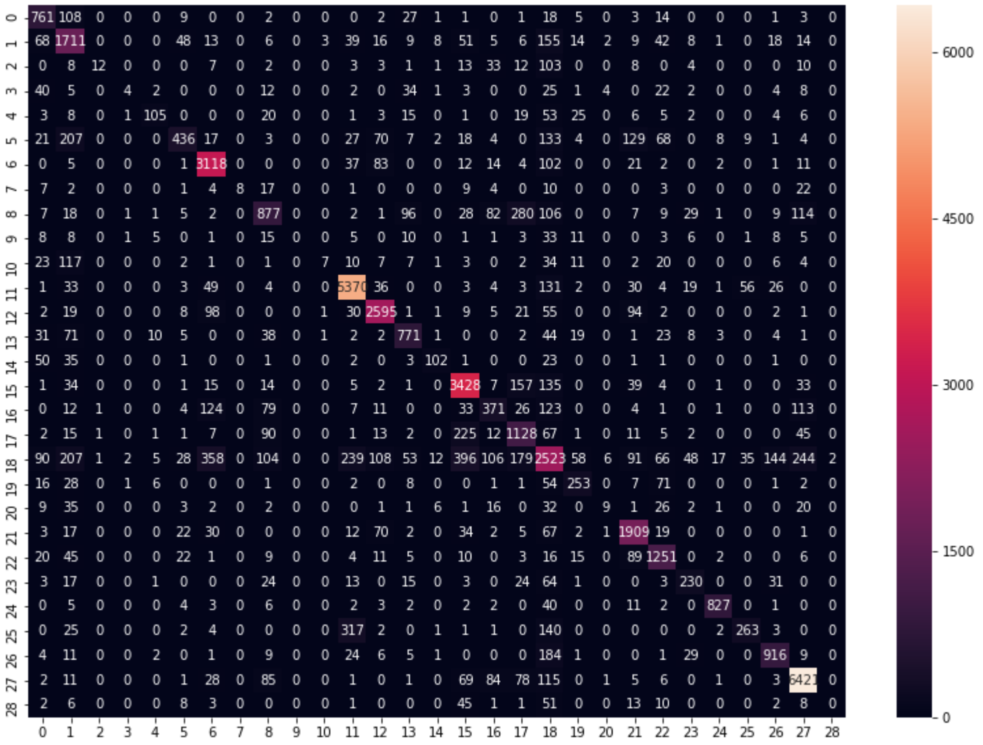


Figure 73: Confusion Matrix of CNN trained on custom FastText embeddings (with skipgram) with lower case descriptions

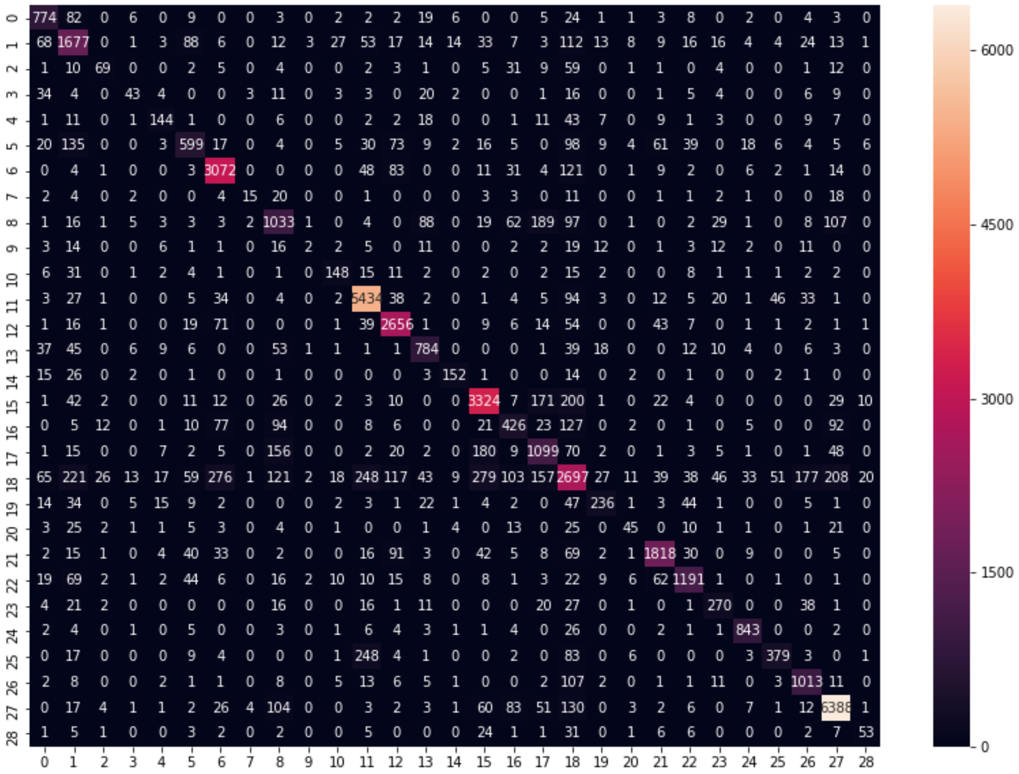
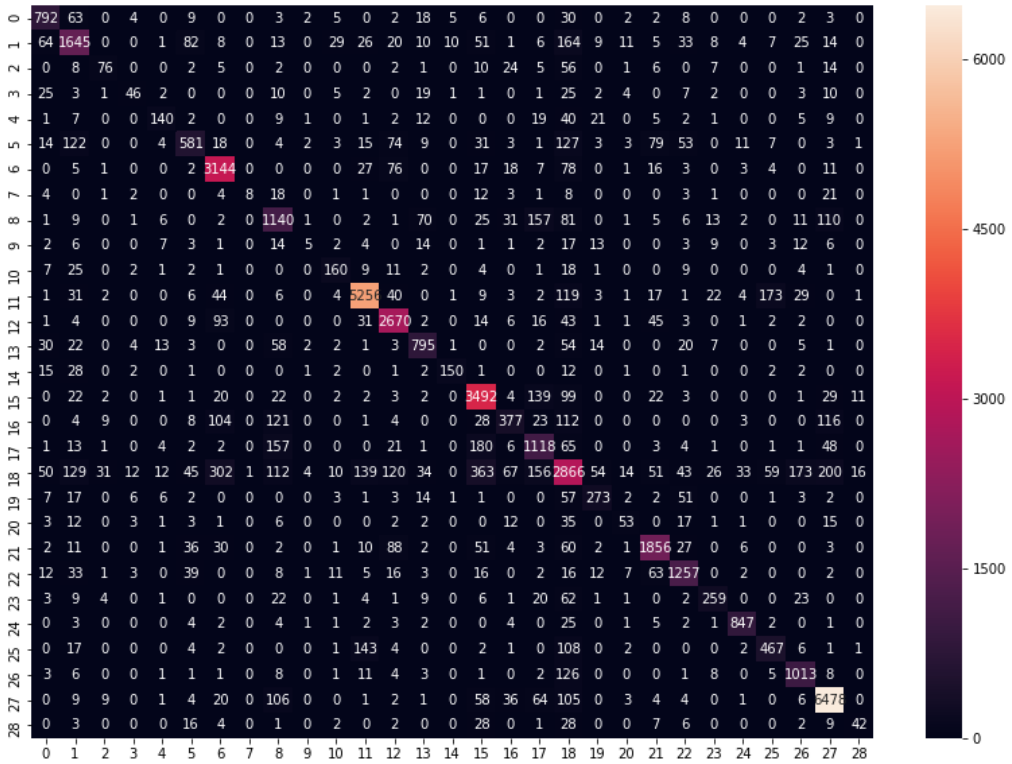
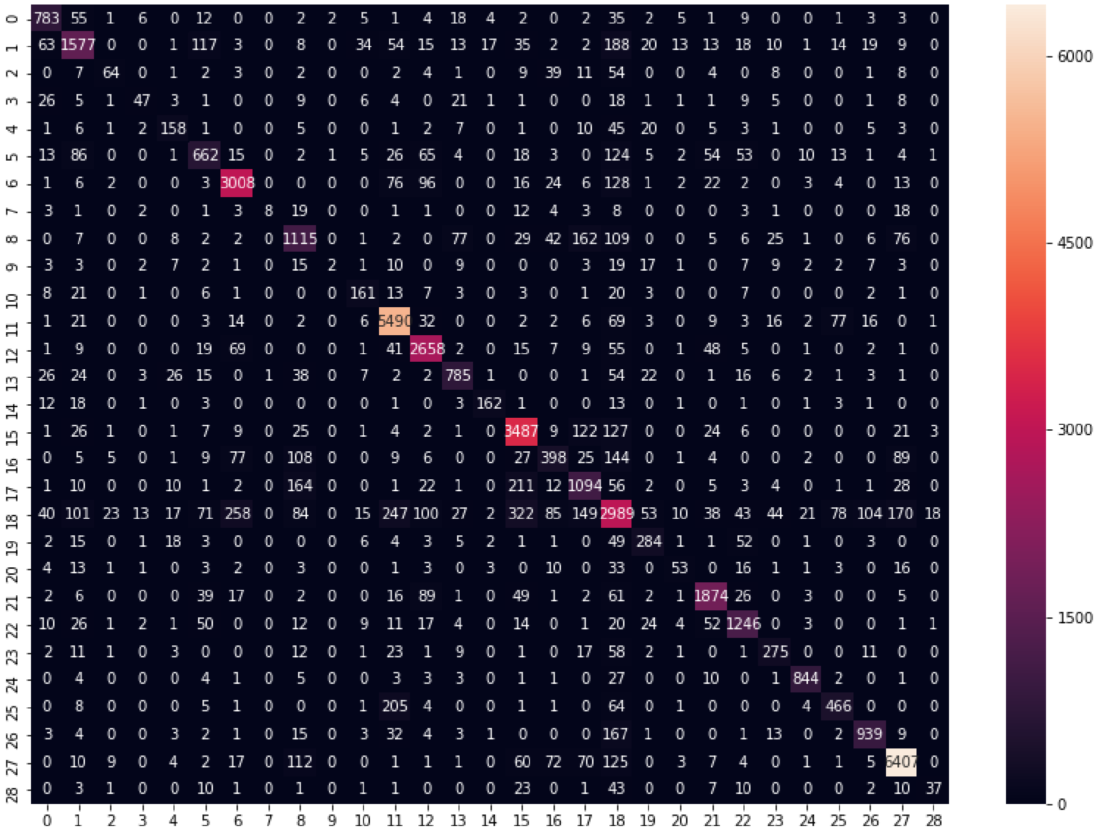


Figure 74: Confusion Matrix of RNN trained on Google's word2vec (Glove) embeddings with lower case descriptions



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	792	63	0	4	0	9	0	0	3	2	5	0	2	18	5	6	0	0	30	0	2	2	8	0	0	0	2	3	0
1	64	1645	0	0	1	82	8	0	13	0	29	26	20	10	10	51	1	6	164	9	11	5	33	8	4	7	25	14	0
2	0	8	76	0	0	2	5	0	2	0	0	0	2	1	0	10	24	5	56	0	1	6	0	7	0	0	1	14	0
3	25	3	1	46	2	0	0	0	10	0	5	2	0	19	1	1	0	1	25	2	4	0	7	2	0	0	3	10	0
4	1	7	0	0	140	2	0	0	9	1	0	1	2	12	0	0	0	19	40	21	0	5	2	1	0	0	5	9	0
5	14	122	0	0	4	581	18	0	4	2	3	15	74	9	0	31	3	1	127	3	3	79	53	0	11	7	0	3	1
6	0	5	1	0	0	2	3144	0	0	0	0	27	76	0	0	17	18	7	78	0	1	16	3	0	3	4	0	11	0
7	4	0	1	2	0	0	4	8	18	0	1	1	0	0	0	12	3	1	8	0	0	0	3	1	0	0	0	21	0
8	1	9	0	1	6	0	2	0	1140	1	0	2	1	70	0	25	31	157	81	0	1	5	6	13	2	0	11	110	0
9	2	6	0	0	7	3	1	0	14	5	2	4	0	14	0	1	1	2	17	13	0	0	3	9	0	3	12	6	0
10	7	25	0	2	1	2	1	0	0	0	160	9	11	2	0	4	0	1	18	1	0	0	9	0	0	0	4	1	0
11	1	31	2	0	0	6	44	0	6	0	4	5256	40	0	1	9	3	2	119	3	1	17	1	22	4	173	29	0	1
12	1	4	0	0	0	9	93	0	0	0	0	31	2670	2	0	14	6	16	43	1	1	45	3	0	1	2	2	0	0
13	30	22	0	4	13	3	0	0	58	2	2	1	3	795	1	0	0	2	54	14	0	0	20	7	0	0	5	1	0
14	15	28	0	2	0	1	0	0	0	1	2	0	1	2	150	1	0	0	12	0	1	0	1	0	0	2	2	0	0
15	0	22	2	0	1	1	20	0	22	0	2	2	3	2	0	3492	4	139	99	0	0	22	3	0	0	0	1	29	11
16	0	4	9	0	0	8	104	0	121	0	0	1	4	0	0	28	377	23	112	0	0	0	0	0	3	0	0	116	0
17	1	13	1	0	4	2	2	0	157	0	0	0	21	1	0	180	6	1118	65	0	0	3	4	1	0	1	1	48	0
18	50	129	31	12	12	45	302	1	112	4	10	139	120	34	0	363	67	156	2866	54	14	51	43	26	33	59	173	200	16
19	7	17	0	6	6	2	0	0	0	0	3	1	3	14	1	1	0	0	57	273	2	2	51	0	0	1	3	2	0
20	3	12	0	3	1	3	1	0	6	0	0	0	2	2	0	0	12	0	35	0	53	0	17	1	1	0	0	15	0
21	2	11	0	0	1	36	30	0	2	0	1	10	88	2	0	51	4	3	60	2	1	1856	27	0	6	0	0	3	0
22	12	33	1	3	0	39	0	0	8	1	11	5	16	3	0	16	0	2	16	12	7	63	1257	0	2	0	0	2	0
23	3	9	4	0	1	0	0	0	22	0	1	4	1	9	0	6	1	20	62	1	1	0	2	259	0	0	23	0	0
24	0	3	0	0	0	4	2	0	4	1	1	2	3	2	0	0	4	0	25	0	1	5	2	1	847	2	0	1	0
25	0	17	0	0	0	4	2	0	0	0	1	143	4	0	0	2	1	0	108	0	2	0	0	0	2	467	6	1	1
26	3	6	0	0	1	1	1	0	8	0	1	11	4	3	0	1	0	2	126	0	0	0	1	8	0	5	1013	8	0
27	0	9	9	0	1	4	20	0	106	0	0	1	2	1	0	58	36	64	105	0	3	4	4	0	1	0	6	6478	0
28	0	3	0	0	0	16	4	0	1	0	2	0	2	0	0	28	0	1	28	0	0	7	6	0	0	0	2	9	42

Figure 75: Confusion Matrix of RNN trained on custom word2vec embeddings (with skipgram) with lower case descriptions



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	783	55	1	6	0	12	0	0	2	2	5	1	4	18	4	2	0	2	35	2	5	1	9	0	0	1	3	3	0
1	63	1577	0	0	1	117	3	0	8	0	34	54	15	13	17	35	2	2	188	20	13	13	18	10	1	14	19	9	0
2	0	7	64	0	1	2	3	0	2	0	0	2	4	1	0	9	39	11	54	0	0	4	0	8	0	0	1	8	0
3	26	5	1	47	3	1	0	0	9	0	6	4	0	21	1	1	0	0	18	1	1	1	9	5	0	0	1	8	0
4	1	6	1	2	158	1	0	0	5	0	0	1	2	7	0	1	0	10	45	20	0	5	3	1	0	0	5	3	0
5	13	86	0	0	1	662	15	0	2	1	5	26	65	4	0	18	3	0	124	5	2	54	53	0	10	13	1	4	1
6	1	6	2	0	0	3	3008	0	0	0	0	76	96	0	0	16	24	6	128	1	2	22	2	0	3	4	0	13	0
7	3	1	0	2	0	1	3	8	19	0	0	1	1	0	0	12	4	3	8	0	0	0	3	1	0	0	0	18	0
8	0	7	0	0	8	2	2	0	1115	0	1	2	0	77	0	29	42	162	109	0	0	5	6	25	1	0	6	76	0
9	3	3	0	2	7	2	1	0	15	2	1	10	0	9	0	0	0	3	19	17	1	0	7	9	2	2	7	3	0
10	8	21	0	1	0	6	1	0	0	0	161	13	7	3	0	3	0	1	20	3	0	0	7	0	0	0	2	1	0
11	1	21	0	0	0	3	14	0	2	0	6	5490	32	0	0	2	2	6	69	3	0	9	3	16	2	77	16	0	1
12	1	9	0	0	0	19	69	0	0	0	1	41	2658	2	0	15	7	9	55	0	1	48	5	0	1	0	2	1	0
13	26	24	0	3	26	15	0	1	38	0	7	2	2	785	1	0	0	1	54	22	0	1	16	6	2	1	3	1	0
14	12	18	0	1	0	3	0	0	0	0	0	1	0	3	162	1	0	0	13	0	1	0	1	0	1	3	1	0	0
15	1	26	1	0	1	7	9	0	25	0	1	4	2	1	0	3487	9	122	127	0	0	24	6	0	0	0	0	21	3
16	0	5	5	0	1	9	77	0	108	0	0	9	6	0	0	27	398	25	144	0	1	4	0	0	2	0	0	89	0
17	1	10	0	0	10	1	2	0	164	0	0	1	22	1	0	211	12	1094	56	2	0	5	3	4	0	1	1	28	0
18	40	101	23	13	17	71	258	0	84	0	15	247	100	27	2	322	85	149	2989	53	10	38	43	44	21	78	104	170	18
19	2	15	0	1	18	3	0	0	0	0	6	4	3	5	2	1	1	0	49	284	1	1	52	0	1	0	3	0	0
20	4	13	1	1	0	3	2	0	3	0	0	1	3	0	3	0	10	0	33	0	53	0	16	1	1	3	0	16	0
21	2	6	0	0	0	39	17	0	2	0	0	16	89	1	0	49	1	2	61	2	1	1874	26	0	3	0	0	5	0
22	10	26	1	2	1	50	0	0	12	0	9	11	17	4	0	14	0	1	20	24	4	52	1246	0	3	0	0	1	1
23	2	11	1	0	3	0	0	0	12	0	1	23	1	9	0	1	0	17	58	2	1	0	1	275	0	0	11	0	0
24	0	4	0	0	0	4	1	0	5	0	0	3	3	3	0	1	1	0	27	0	0	10	0	1	844	2	0	1	0
25	0	8	0	0	0	5	1	0	0	0	1	205	4	0	0	1	1	0	64	0	1	0	0	0	4	466	0	0	0
26	3	4	0	0	3	2	1	0	15	0	3	32	4	3	1	0	0	0	167	1	0	0	1	13	0	2	939	9	0
27	0	10	9	0	4	2	17	0	112	0	0	1	1	1	0	60	72	70	125	0	3	7	4	0	1	1	5	6407	0
28	0	3	1	0	0	10	1	0	1	0	1	1	0	0	0	23	0	1	43	0	0	7	10	0	0	0	2	10	37

Figure 76: Confusion Matrix of RNN trained on custom FastText embeddings (with skipgram) with lower case descriptions