**Classification for the Detection of Opinion Spam**  
***Data Mining 2020, Utrecht University***  
*Assignment 2*

Bentis Nikolaos (6662889) Polos Markos (6943721) Tsogias Ioannis 6966985)

1. **Introduction**

With the advent of social networking sites, opinion-mining applications have attracted the interest of the online community on review sites to know about products for their purchase decisions. It has become an essential part of their success for e-commerce businesses to empower their end customers to write feedback or reviews about the products or services that they have utilized. There is also growing fad of appraising online services and products that the customer has experienced. Such reviews provide vital sources of information on these products or services. This information is utilized by the future potential customers before deciding on purchase of new products or services. These opinions or reviews are also exploited by marketers to find out the drawbacks of their own products or services and alternatively to find the vital information related to their competitor’s products or services. This in turn allows to identify weaknesses or strengths of products. **[1]**

For example, if one wants to buy a product, he/she typically goes to a review site to read some reviews of the product. If most reviews are positive, one is likely to buy the product. If most reviews are negative, one will almost certainly not buy it. Positive opinions can result in significant financial gains and/or fames for businesses, organizations and individuals. This, unfortunately, gives strong incentives for opinion spamming.

Opinion spamming refers to "illegal" activities (e.g. writing fake reviews) that try to deliberately mislead readers or automated opinion mining and sentiment analysis systems by giving undeserving positive opinions to some target entities in order to promote the entities and/or by giving false negative opinions to some other entities in order to damage their reputation. Opinion spam comes in many forms, e.g., fake reviews, fake comments, fake blogs, fake social network postings, deceptions, and deceptive messages. Hence, the increasing trend of posting spam (fake) reviews to promote the target products or defame the specific brands of competitors led to the emergence of Opinion Spam detection and classification as a hot issue in the community of opinion mining and sentiment analysis.

In this case, we approach the issue of Opinion Spam detection by using different classifiers as the Multinomial Naive Bayes, Regularized Logistic Regression, Classification Trees and Random forests whose performance will be estimated with and without the addition of bigram features. As for the data, we analyze fake and genuine hotel reviews that have been collected by Myle Ott and others **[2][ 3]** and focus on the negative reviews trying to discriminate between truthful and deceptive reviews. The genuine reviews have been collected from several popular online review communities and the fake reviews have been obtained from Mechanical Turk.

1. **Describing the Data**

The dataset that this project was based on was created and published by Ott **[2][3]** Originally the dataset consisted 1600 reviews of hotels from the Chicago area, of which 800 were positive and 800 were negative. In more detail, the reviews are focused only on the 20 most popular hotels in the area. In each category 400 were genuine user reviews extracted from TripAdvisor and the remaining 400 were false reviews created with the use of Amazon Mechanical Turk (AMT), in which people were asked to create these fake reviews for financial compensation.

A lot of effort was made by the creators of the dataset to make sure that the deceptive reviews were similar to deceptive ones that people wrote in the real world to hurt or benefit businesses. Each person writing a fake review could do it only once, to ensure that there are not “bots” that were created to do this and earn money and there was time limit restriction. After gathering the required number of fake reviews, these were used to select real reviews that had similar length. At the end, they managed to construct a balanced dataset in every aspect, as for each hotel there were 40 real and 40 deceptive reviews and half of each category were positive and the rest negative.

For our purposes the part of the dataset containing positive reviews was not used, as our focus was on the classification between real and deceptive negative reviews. Out of the 800 negative reviews, 640 of them were used for training and hyper-parameter tuning and the reaming for testing.

1. **Setup and Data Preprocessing**

**3.1 Data Preprocessing**

In any machine learning task, cleaning or preprocessing the data is as important as model building, if not more, and when it comes to unstructured data like text, this process carries even higher importance. In this particular case, we concluded that several preprocessing steps should be performed in order to provide to the models the necessary amount of information out of the text reviews.

Before preprocessing the data, all .txt files of the dataset were inserted into a single .csv file with their “0”/”1” labels as in “0” meaning that the review is not fake and “1” that the review is actually fake/spam. The creation of a more efficient/accessible structure for the data allows for a more efficient work during the preprocessing and experimenting phase.

The first stage of the preprocessing phase is ***lowercasing***, a very common text preprocessing technique that transforms all capital letters into lowercase letters. Lowercasing was applied to the dataset in order to make the dictionary of words smaller and give the words the weight that it suits them, for example we avoid having “Playing” and “playing” as to separate words. However, we observed that in many cases negative reviews have fully capitalized sentences which may show frustration/anger (e.g. "THIS IS HORRIBLE"). So, lowercasing every single word might lead to losing some information that might have given a lead if the review is fake or not. To avoid that, a custom lowercasing function that lowercases only the words whose first character is capital, was implemented.

Secondly, python’s function ***fix()*** of the library ***Contractions*** was used in order to keep the structure of the sentences formal, and assign the correct counts of words in the dictionary. Contractions affect the quality of the Reviews and some words could be accounted multiple times instead of the correct number. For example, "Y'all" should be two tokens when examining the review, with the correct form to be "You all", which is done by the aforementioned function.

The next step is to ***remove punctuations*** from the texts. For this purpose, an already made list from the library ***String*** will be used which contains: “"!#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~”. However, we decided to not remove "!" for the dataset because in many reviews the number of "!" next to words can carry some information. Essentially, punctuations are removed for the same reason as before, which is to treat "word" and "word." as the same. This process will lower the number tokens in the dictionary and will help assign the proper weights to each word (e.g. "word"|"word." has the same meaning in both cases). An examination of the dataset showed that no emoticons like ":)" were used in the reviews, so there is no loss of information by removing punctuations. In case there were emoticons, the proper way would be to first "translate" them with their meaning and then removing all the remaining punctuations.

What should follow, is the ***removal of stop*** ***words*** from the Reviews. Stop words (such as “the”, “a”, “an”, “in”) usually do not carry any meaning and account for a large number of words[could find a reference]. By removing them we have less instances in our dictionary and the training of our algorithms will take less time as less words will be examined each time. Library NLTK[reference] has a list of stop words which is the common list used in most text preprocessing tasks.

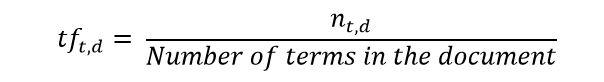
In order to make our dictionary even smaller and the models even faster we want to ***remove frequent words*** that do not provide information, these words should be frequent in both real and fake reviews. This distinction is importnat, because with a balanced dataset such as the one we are using, we can't blindly remove frequent words as they could be frequent only at one of the two labels. For that purpose, a custom function is created that searches for frequent words in the Reviews of each label separately, then the lists of the 20 most frequent words from each case are being compared and the words appearing in both cases are being removed. This indicates that the remaining words from each list will be features with high importance for the label they belong.

The preprocessing steps mentioned above, were applied in all cases. However, we wanted to see how all models would perform if lemmatization or stemming was applied in the texts. ***Stemming [reference]*** is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form. For example, if there are two words in the corpus “walks” and “walking”, then stemming will stem the suffix to make them “walk”. ***Lemmatization [ref]*** is similar to stemming in reducing inflected words to their word stem but differs in the way that it makes sure the root word (also called as lemma) belongs to the language. For Stemming, PorterStemmer() from NLTK was used and for Lemmatization we used the WordNetLemmatizer() again from the NLTK library. The cases where the performance of a model is better under one of these settings will be presented in the Results section.

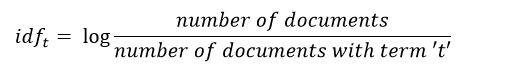
Generally, there are also other pre-processing options. One option is removing Rare words, but rare words could carry meaning and we could also dictate that through the algorithm initialization. Additionally, there were no noise in the data like HTML tags or URLS that should be removed.

**3.2 Splitting into Test/Train sets & Vectorizing**

The next step is the feature extraction phase. Here we used the TFIDFVectorizer from sklearn **[4]** so it normalizes the counts of the features that appear as common. With the TFIDFVectorizer the score of each word increases proportionally to count, as seen at the term frequency (TF) part, but is inversely proportional to frequency of the word in the corpus; that is the inverse document frequency (IDF) part. The following equations describe the way that the scores of each term are calculated.



*Figure 1: Term Frequency part of TfiDF Vectorizer*



*Figure 2: Document Frequency part of TfiDF Vectorizer*

TF_IDF formula

*Figure 3: TfIDf equation*

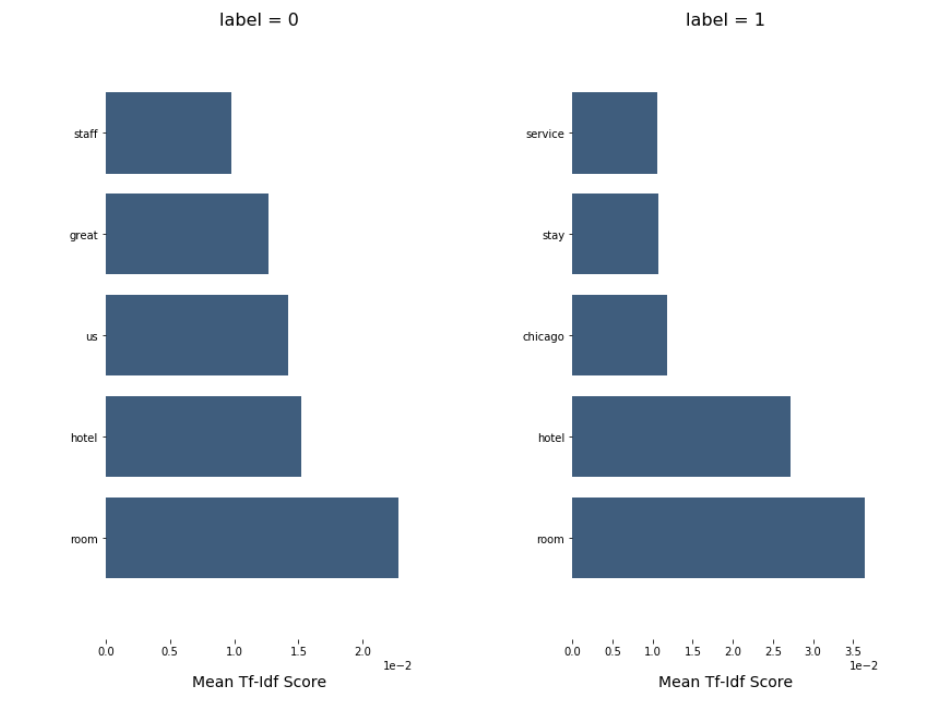
The inverse document frequency adjusts for the fact that some words appear more frequently in general. In more detail, even though in the negative reviews’ words like “bad” and “location” appear often in documents, they appear often in all documents. Hence, they don’t provide a lot of information about the features that make a certain document unique. If any unusual words like “crepuscular” or “aspergillum” appear multiple times in a document provide more information because words like these will most likely not appear frequently in the corpus. “crepuscular” tells us a lot more about what the document is about than “bad” and “location”.

Before vectorizing though, we split the dataset into train and test set. To do that we use function train\_test\_split() from sklearn. Since we use folds 1-4 (640 reviews) for training and hyper-parameter tuning and Fold 5 (160 reviews) to estimate the performance of the classifiers, we want the test set to be 20% of the whole data frame and set shuffle = False in order to take the last 20% of the dataset as a test set.

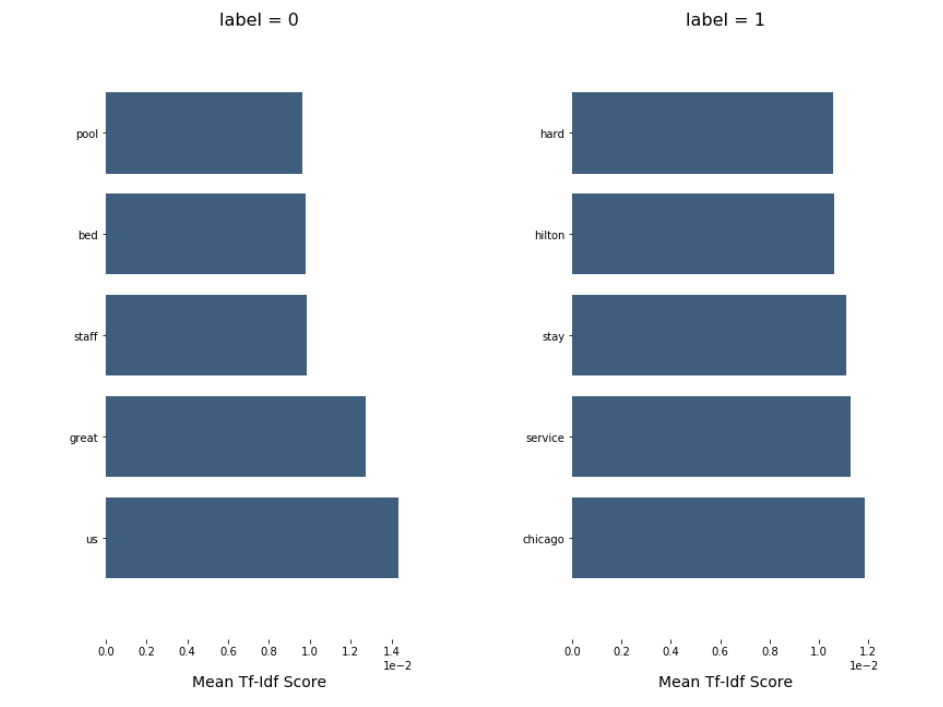
After splitting, we vectorize our data, setting lowercase=False since we created our own function for this purpose, ngram\_range=(1, 1) for the unigram models and ngram\_range=(1, 2) for the bigram models.

With the use of the TfiDfVectorizer on the training set and by examining the list of terms for the set of fake and real reviews separately we can distinguish the most important words/terms for each class that point to that class direction. First, we are going to see the 5 most important features per class, without removing the frequent features that appear in both classes, as we perform in our preprocessing steps. Then, we are going to perform the same analyses but with the frequent features removed. Finally, we are going to repeat the analysis with the addition of lemmatization.

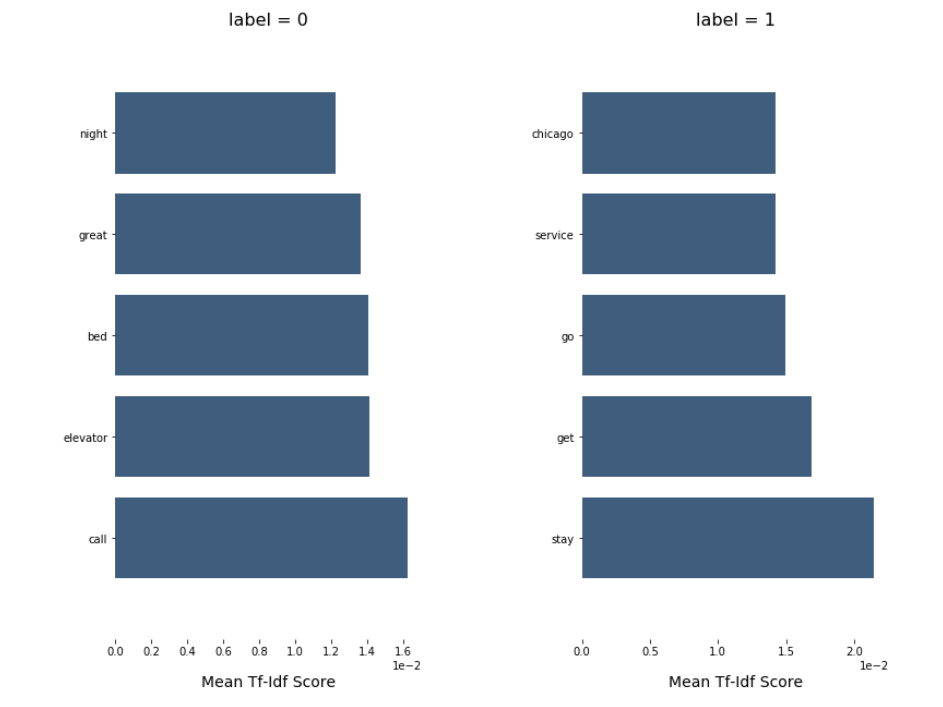
As seen at the three following figures, each preprocessing step is important. First the most important features for both classes are the same and their removal is beneficial for the classification process. Additionally, by using Lemmatization some terms increased their importance for each class. These lists provide some information even when only 5 features are seen for each class. For “label = 0” which is the “real” class the important words are used to describe the hotel/experience, such as “great”, “elevator” and “staff”, but for the “label = 1”, the “fake” class the important terms state where the reviewer stayed, examples are “Chicago” , “stay”, “go” and “hilton”.



*Figure 4: Most important Features per Class - simple case*



*Figure 5: Most important Features per class - Removed Frequent*



*Figure 6: Most important Features per class - Removed Frequent + Lemma*

**3.3 Setting up the models**

**3.3.1. Multinomial naive Bayes**

The first model trained was Naive Bayes. We used the multinomial NB implementation of sklearn for the model. Naive Bayes is a very simple model and no hyper-parameter tuning was used during its set up phase. Instead, we decided to explore its performance with a different size of features. Having acquired the features after the vectorization phase, we computed the chi-squared **[5]** values of each term using the method provided by sklearn and then proceeded to select only a number of them based on their scores, starting from the best features and using the “selectKBest” method from sklearn.

In our experiments to find out the optimal K value of the best features, we had different options for the unigram and the bigram models, as the number of features is different in each model. In unigram, the word “hello” is one term, but in bigram we have “hello|say” and every other combination. For the unigram model and depending on the preprocessing steps the number of features is close to 6500, but for the bigram model this number is close to 45000. The values of k examined for the unigram model are between 100 and 3000 features, for the bigram these are between 500 and 10000. For each case after fitting (fit()) the model, we then used the test set to predict() the classes and then performed the following statistical tests: accuracy, precision, recall, f1-score and confusion matrix.

**3.3.2. Regularized logistic regression**

In order to find the best hyperparameters for the ***Regularized Logistic Regression model***, a grid search among several values must be performed. Firstly, we must choose between ‘L1’ and ‘L2’ penalty. A linear regression model that implements L1 norm for regularization is called Lasso Regression, and one that implements (squared) L2 norm for regularization is called Ridge Regression. Also, a parameter that needs to be tuned is C which corresponds to the Inverse of regularization strength; must be a positive float (C = 1/λ). That is a control variable that retains strength modification of Regularization by being inversely positioned to the Lambda regulator. The relationship would be that lowering C - would strengthen the Lambda regulator.

For this task, the function *LogisticRegression(solver='liblinear', penalty=p,C=c)* from sklearn was used. Solver was set at ‘*liblinear’* as according to the documentation, ‘*liblinear’* is a good choice for small datasets. Next, p and c are lists that contain [‘l1’,’l2’] and [0.01,0.1,1,10,100] respectively.

After defining the possible values of the algorithm’s hyperparameters, a for-loop was implemented in order to run and keep track of the performance of all models generated through every possible combination of the hyperparameters. In each iteration the model was defined with the aforementioned function. Then, *fit()* and *predict()* (also from sklearn) were used to train/fit the linear model and get its predictions on the test set. Lastly, the evaluation metrics (Accuracy, Recall, Precision, and F1- Score) were saved with the confusion matrix for each model. Inside this loop, both the uni-gram and bi-gram models were tuned.

**3.3.3. Classification trees**

Classification trees are an excellent way to classify classes as it gives us the ability to actually understand the logic behind each decision classification. As in the previous models, a grid search method will be used to find the best hyperparameters for this method. First a dictionary of params will be created with all the possible values that will be tested. Next, the classifier and the parameters are given to the grid search. Finally the top parameters are given that resulted with the best score. The same process is repeated for both unigram and bigram models.

There are several parameters that were used to calculate the best score. The param *max\_depth* is used to get the maximum depth of the tree. The *min\_samples\_split* is the minimum number of samples required to split an internal node and finally the *min\_samples\_leaf* is the minimum number of samples that are required so it can be a leaf node. A wide range of values were used for these parameters. For the *max\_depth* the values [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None] were chosen along with [2,5,10,15,20] for the *min\_samples\_split* and [1,2,4,8,10,20] for the *min\_samples\_leaf.*

**3.3.4. Random Forests**

Random forests consist of a large number of individual classification trees. The tree with the best score will be selected for the model classification. The parameters that will be used in this model are going to be the same as in the classification trees but in order to determine how many trees the Random Forest will consist of, the *n\_estimators* param is added. The number of trees that will be compared are a selection of numbers [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000] where they will be combined with the classification tree’s params. For calculating the best score we follow the same logic as in the classification tree.

1. **Results**
   1. **Multinomial naive Bayes**

On multinomial naïve Bayes many models were tested depending on both the preprocessing steps, described on section 3.1, and the number “k” as described on section 3.3. For starters, we tested the value of k between 100 and 3000 for unigram and of k between 500 and 10000 for bigram, with the standard preprocessing step for both for unigram and bigram. The best model out of the unigram models had k equal to 500 with an accuracy of 0.8312 and a f1-score of 0.844. The best bigram had a k equal to 5000 and succeeded better results with an accuracy of 0.8625 and a f1-score of 0.8721.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Unigram model, k = 500 | 0.8312 | 0.7849 | 0.9125 | 0.844 |
| Bigram model,  K = 5000 | 0.8625 | 0.81521 | 0.937 | 0.8721 |

*Table 1: Performance metrics for Multinomial Naïve Bayes, Normal Preprocessing Settings*

The confusion matrices for the models are placed below:

|  |  |
| --- | --- |
| **60** | **20** |
| **7** | **73** |

*Table 4.2.2. Confusion Matrix for Unigram Model, k =500*

|  |  |
| --- | --- |
| **63** | **17** |
| **5** | **75** |

*Table 4.2.3. Confusion Matrix for Bigram Model, k = 500*

**Experimenting**

Additionally, as mention on section 3.1 we experimented with more preprocessing options after our basic ones. We tried adding Lemmatization or stemming to the end of the preprocessing steps and then examined each one of them with different values of k as before. For lemmatization, the best unigram model used the 300 best features, it had an accuracy of 0.8375 and a f1-score of 0.841. The best bigram model used the 3000 best features, it had an accuracy of 0.85625 and a f1-score of 0.865. For Stemming, the best unigram model had k equal to 500 with an accuracy of 0.8375 and a f1-score of 0.847. The stemmed bigram model had k equal to 500 with an accuracy of 0.8625 and a f1-score of 0.8641.

After all the experiments we observed that the bigram models exceeded the performance of the unigram models constantly. In addition, the best Naïve Bayes model is the one without stemming or lemmatization, so these preprocessing steps for the Naïve Bayes models are unnecessary.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Unigram model,  Lemmatization, k= 300 | 0.8375 | 0.821 | 0.8625 | 0.841 |
| Bigram model,  Lemmatization,  K = 3000 | 0.85625 | 0.813 | 0.925 | 0.865 |

*Table 4.2.1. Performance metrics for Multinomial Naïve Bayes, Normal Preprocessing Settings +lemmatization*

Here, in **Tables 4.2.2**. and **4.2.3.**, are the confusion matrices for the models described in **Table 4.2.1.**

|  |  |
| --- | --- |
| **65** | **15** |
| **11** | **69** |

*Table 4.2.2. Confusion Matrix for Unigram Model, lemmatization and k = 300.*

|  |  |
| --- | --- |
| **63** | **17** |
| **6** | **74** |

*Table 4.2.3. Confusion Matrix for Bigram Model, lemmatization and k = 3000.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Unigram model, stemming,  k = 500 | 0.8375 | 0.8 | 0.9 | 0.847 |
| Bigram model,  Stemming,  K = 500 | 0.8625 | 0.8536 | 0.875 | 0.8641 |

*Table 4.2.1. Performance metrics for Multinomial Naïve Bayes, Normal Preprocessing Settings +stemming*

Here, in Tables 4.2.2. and 4.2.3., are the confusion matrices for the models described in Table 4.2.1.

|  |  |
| --- | --- |
| **62** | **18** |
| **8** | **72** |

*Table 4.2.2. Confusion Matrix for Unigram Model, stemming and k = 500*

|  |  |
| --- | --- |
| **68** | **12** |
| **10** | **70** |

*Table 4.2.3. Confusion Matrix for Bigram Model, stemming and k = 500.*

**4.2. Regularized logistic regression**

As stated in the previous section, all possible combinations for the hyperparameters of the model were tested and the best results for the basic preprocessing settings are presented in Table 4.2.1. As it is showcased by the Table below, the best unigram model was tuned with penalty = ‘L2’ and C = 1 and achieved 0.8758 F1 score, while the best bigram model has penalty = ‘L2’ and C = 10 with its F1 score to be equal to 0.8641.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Unigram model, penalty = ‘L2’, C = 1 | 0.8812 | 0.9178 | 0.8375 | 0.8758 |
| Bigram model,  Penalty = ‘L2’  C = 10 | 0.8625 | 0.8536 | 0.875 | 0.8641 |

*Table 4.2.1. Performance metrics for Regularized Logistic Regression, Normal Preprocessing Settings*

Here, in Tables 4.2.2. and 4.2.3., are the confusion matrices for the models described in Table 4.2.1.

|  |  |
| --- | --- |
| **74** | **6** |
| **13** | **67** |

*Table 4.2.2. Confusion Matrix for Unigram Model, penalty='L2', C=1.*

|  |  |
| --- | --- |
| **68** | **12** |
| **10** | **70** |

*Table 4.2.3. Confusion Matrix for Bigram Model, penalty='L2', C=10.*

Since we are looking for the differences in the performance when adding bigram features, maybe it would be useful to see the respective bigram for the best Unigram and vice versa. That is, we will now look into the performance of the Unigram with penalty = ‘L2’ and C = 10 and the performance of the Bigram with penalty = ‘L2’ and C = 1. Their results are presented in Table 4.2.4 and confusion matrices at Table 4.2.5. and 4.2.6.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Bigram model, penalty = ‘L2’, C = 1 | 0.85 | 0.83333 | 0.875 | 0.8536 |
| Unigram model,  Penalty = ‘L2’  C = 10 | 0.85 | 0.8888 | 0.8 | 0.8421 |

*Table 4.2.4. Performance metrics for Regularized Logistic Regression, Normal Preprocessing Settings, Respective models.*

|  |  |
| --- | --- |
| **66** | **14** |
| **10** | **70** |

*Table 4.2.5. Confusion Matrix for Bigram Model, penalty='L2', C=1.*

|  |  |
| --- | --- |
| **72** | **8** |
| **16** | **64** |

*Table 4.2.6. Confusion Matrix for Unigram Model, penalty='L2', C=10.*

These results indicate that in the case of Regularized Logistic Regression, adding bigram features instead of just using unigrams does not improve the performance of the model. In terms of Accuracy and F1 score, the best Unigram model (penalty = ‘L2’, C = 1) outperforms all the others.

**Experimenting**

What was interestingly observed when making several trials on the way that certain preprocessing steps affect the performance of the model, was that both Unigram and Bigram models increased their performance when Lemmatization was applied to the data before training the model. For a more detailed examination, the results we got from this setting are presented in Tables 4.2.7, 4.2.8, 4.2.9.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Unigram model, penalty = ‘L2’, C = 1 | 0.8875 | 0.9189 | 0.85 | 0.8831 |
| Bigram model,  Penalty = ‘L2’  C = 10 | 0.8812 | 0.8860 | 0.875 | 0.8805 |

*Table 4.2.7. Performance metrics for Regularized Logistic Regression, Normal Preprocessing Settings + Lemmatization.*

|  |  |
| --- | --- |
| **74** | **6** |
| **12** | **68** |

*Table 4.2.8. Confusion Matrix for Unigram Model, penalty='L2', C=1 + Lemmatization.*

|  |  |
| --- | --- |
| **71** | **9** |
| **10** | **70** |

*Table 4.2.9. Confusion Matrix for Bigram Model, penalty='L2', C=10 + Lemmatization.*

Again, it would be insightful to examine how the respective to the best models perform in order to investigate whether adding Lemmatization in preprocessing, contributes in making the addition of bigram features useful.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Bigram model, penalty = ‘L2’, C = 1 | 0.875 | 0.875 | 0.875 | 0.875 |
| Unigram model,  Penalty = ‘L2’  C = 10 | 0.86875 | 0.8933 | 0.8375 | 0.8645 |

*Table 4.2.7. Performance metrics for Regularized Logistic Regression, Respective models, Normal Preprocessing Settings + Lemmatization.*

|  |  |
| --- | --- |
| **70** | **10** |
| **10** | **70** |

*Table 4.2.8. Confusion Matrix for Bigram Model, Respective Model, penalty='L2', C=1 + Lemmatization.*

|  |  |
| --- | --- |
| **72** | **8** |
| **13** | **67** |

*Table 4.2.9. Confusion Matrix for Unigram Model, Respective Model, penalty='L2', C=10 + Lemmatization.*

Once again, it is proven that adding Bigram features instead of only using Unigram features does not improve the performance of the Regularized Logistic Regression model since the best Unigram model with penalty = ‘L2’, C = 1 when Lemmatization has been applied, outperforms all others in terms of Accuracy and F1 score.

**4.3. Classification trees**

Continuing with the same logic as in the previous sections, the best results with their corresponding parameters will be presented in the following tables for both the unigram and bigram model.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Bigram model, max\_depth=60, min\_samples\_leaf=4, min\_samples\_split=20 | 0.56875 | 0.57 | 0.57 | 0.57 |
| Unigram model,  max\_depth=60, min\_samples\_leaf=4, min\_samples\_split=20 | 0.625 | 0.63 | 0.62 | 0.62 |

*Table 4.3.1. Performance metrics for Classification trees, Normal Preprocessing Settings*

|  |  |
| --- | --- |
| **45** | **35** |
| **34** | **46** |

*Table 4.3.2. Confusion Matrix for Bigram Model*

|  |  |
| --- | --- |
| **52** | **28** |
| **32** | **48** |

*Table 4.3.3. Confusion Matrix for Unigram Model*

**Experimenting**

As it is done before, the models will be compared first by adding the Lemmatization to our preprocess settings and finally adding the Stemming setting.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Bigram model, max\_depth=20, min\_samples\_leaf=20, min\_samples\_split=10 | 0.70625 | 0.71 | 0.71 | 0.71 |
| Unigram model,  max\_depth=50, min\_samples\_leaf=8, min\_samples\_split=5 | 0.6875 | 0.69 | 0.69 | 0.69 |

*Table 4.3.4. Performance metrics for Classification trees, Normal Preprocessing Settings + Lemmatization.*

|  |  |
| --- | --- |
| **57** | **23** |
| **24** | **56** |

*Table 4.3.5. Confusion Matrix for Bigram Model + Lemmatization*

|  |  |
| --- | --- |
| **61** | **19** |
| **31** | **49** |

*Table 4.3.6. Confusion Matrix for Unigram Model + Lemmatization*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Bigram model, max\_depth=50, min\_samples\_leaf=8, min\_samples\_split=5 | 0.63125 | 0.63 | 0.63 | 0.63 |
| Unigram model,  max\_depth=10, min\_samples\_leaf=20, min\_samples\_split=5 | 0.6625 | 0.66 | 0.66 | 0.66 |

*Table 4.3.7. Performance metrics for Classification trees, Normal Preprocessing Settings + Steaming.*

|  |  |
| --- | --- |
| **49** | **31** |
| **28** | **52** |

*Table 4.3.8. Confusion Matrix for Bigram Model + Steaming*

|  |  |
| --- | --- |
| **49** | **31** |
| **23** | **57** |

*Table 4.3.9. Confusion Matrix for Unigram Model +Steaming*

**4.4. Random Forests**

In Table 4.4.1. the results of the Random Forests with Normal Preprocessing are shown for both biagram and unigram models. The biagram model is slightly better than the unigram in terms of accuracy with the best params being max\_depth=50, min\_samples\_leaf=4, min\_samples\_split=8 and n\_estimators=2000.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Bigram model, max\_depth=50, min\_samples\_leaf=4, min\_samples\_split=8,n\_estimators=2000 | 0.83125 | 0.84 | 0.83 | 0.83 |
| Unigram model,  max\_depth=90, min\_samples\_leaf=3, min\_samples\_split=12, n\_estimators=1000 | 0.825 | 0.83 | 0.82 | 0.82 |

*Table 4.4.1. Performance metrics for Random forests, Normal Preprocessing Settings.*

The confusion matrices for the models are placed below:

|  |  |
| --- | --- |
| **72** | **8** |
| **19** | **61** |

*Table 4.4.2. Confusion Matrix for Bigram Model*

|  |  |
| --- | --- |
| **68** | **12** |
| **16** | **64** |

*Table 4.4.3. Confusion Matrix for Unigram Model*

**Experimenting**

As done previously, the preprocessing settings were changed and compared first by adding the Lemmatization and then adding Streaming. By comparing the two tables 4.4.4. and 4.4.7., it is shown that the best accuracy is achieved by adding lemmatization to the preprocessing settings in the biagram model, with accuracy 0.875.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Bigram model, max\_depth=60, min\_samples\_leaf=3, min\_samples\_split=12, n\_estimators=1000 | 0.875 | 0.88 | 0.88 | 0.87 |
| Unigram model,  max\_depth=100, min\_samples\_leaf=5, min\_samples\_split=8, n\_estimators=1000 | 0.84375 | 0.85 | 0.84 | 0.84 |

*Table 4.4.4. Performance metrics for Random Forest, Normal Preprocessing Settings + Lemmatization.*

|  |  |
| --- | --- |
| **75** | **5** |
| **15** | **65** |

*Table 4.4.5. Confusion Matrix for Bigram Model + Lemmatization*

|  |  |
| --- | --- |
| **71** | **9** |
| **16** | **64** |

*Table 4.4.6. Confusion Matrix for Unigram Model +Lemmatization*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **Precision** | **Recall** | **F1 score** |
| Bigram model, max\_depth=90, min\_samples\_leaf=3, min\_samples\_split=8, n\_estimators=2000 | 0.825 | 0.83 | 0.82 | 0.82 |
| Unigram model,  max\_depth=70, min\_samples\_leaf=4, min\_samples\_split=10, n\_estimators=1000 | 0.8375 | 0.84 | 0.84 | 0.84 |

*Table 4.4.7. Performance metrics for Random Forests, Normal Preprocessing Settings + Steaming.*

|  |  |
| --- | --- |
| **71** | **9** |
| **19** | **61** |

*Table 4.4.8. Confusion Matrix for Bigram Model + Steaming*

|  |  |
| --- | --- |
| **72** | **8** |
| **18** | **62** |

*Table 4.4.9. Confusion Matrix for Unigram Model +Steaming*

1. **Conclusions**

1. How does the performance of the generative linear model (multinomial naïve Bayes) compare to the discriminative linear model (regularized logistic regression)?

2. Is the random forest able to improve on the performance of the linear classifiers?

3. Does performance improve by adding bigram features, instead of using just unigrams?

* 1. **Mc Nemar’s Test**

**References**

1. Patel, Rinki, and Priyank Thakkar. "Opinion Spam Detection Using Feature Selection." Computational Intelligence and Communication Networks (CICN), 2014 International Conference on. IEEE, 2014.
2. Myle Ott, Yejin Choi, Claire Cardie and Jeffrey T. Hancock, Finding deceptive opinion spam by any stretch of the imagination. Proceedings of the 49th meeting of the association for computational linguistics, pp.309-319, 2011.
3. Myle Ott, Claire Cardie and Jeffrey T. Hancock, Negative deceptive opinion spam. Proceedings of NAACL-HLT 2013, pp. 497-501, 2013.
4. <https://scikit-learn.org/stable/>
5. https://en.wikipedia.org/wiki/Chi-squared\_test

Patel, Rinki, and Priyank Thakkar. "Opinion Spam Detection Using

Feature Selection." Computational Intelligence and Communication

Networks (CICN), 2014 International Conference on. IEEE, 2014.