Bilkent University
Spring 2019
CS 464 -Machine Learning Project

HATE SPEECH DETECTOR

Group 13

Nurefşan Müsevitoğlu - 21503062
Yasin Balcancı - 21501109
Semih Teker - 21300964
Nejdet Balkır Göka - 21400551
Muammer Tan - 21400967

1. Introduction

Our aim on this project is to determine the intention of writing a tweet whether it contains a hate speech or not. Tweets are written in natural language and also they include shortcuts. Therefore, the models should understand the language used in each post and draw a conclusion from them. Additionally, since words could have different meanings depending on the context they are used, the model should understand the meaning of the posts not just by looking at each word but by analyzing it in the context.

We have used a dataset used in a paper which includes 85,966 English tweets: 32,119 labeled as hateful and 53,851 labeled as normal [1]. To use this database, we mailed the author and got the permission of the author.

For the first part of the preprocessing, we clean the raw data in the dataset, and for the second part which is feature extraction, 3 methods have been used: Pipeline, Word to Vector (Word2Vec), and Bag of Words (BoW). Also we have tried 9 different algorithms to train and test the data:

- Naive Bayes Classification
- Support Vector Machine (SVM)
- Logistic Regression
- K Nearest Neighbor (kNN)
- Decision Tree Classification
- Random Forest Classification
- Gradient Boosting Classification
- XGBoost
- Recurrent Neural Network (RNN)

The results after applying these methods have been compared to see which algorithm works best for this type of problem.

2. Problem description

Social media provides the chance to people of explaining themselves to whole world. When they want to express their thoughts, emotions, ideas etc. they use social media. In 2019, number of social media users is around 2.77 billion worldwide. [2] Active Twitter users around the world is 330 million. [3]

Allowing people to express themselves freely may result in offensive behaviors to other people. It is reported that posts including hate speech in social media are increasing every year [4]. Social media companies' procedures on hate speech show difference in detail but generally they are based on complaints and deleted manually. Twitter has the lowest removal rate on hate speech. [5].

We want to implement hate speech detection that helps social media companies' to handle them before the post without any complaint. Best result of this procedure will be people that may offended from the post will not see them and get offended or hurt.

3. Methods

We have 85,966 tweets in our dataset. However, these tweets contain upper case letters, mentions and hashtags, url addresses, punctuations and emoji characters, numericals, stopwords and slangs.

For the first part of the preprocessing, we have cleaned the tweets: change the text to lowercase, remove the mentions and hashtags, remove the url addresses, remove the punctuations, emoji characters and bad characters, remove the numericals, remove the stopwords, remove the words that are not in the english dictionary to avoid slangs, clean the unnecessary spaces in the tweets. After these processes, some of the tweets might have get empty and these empty tweets have been dropped from the dataset and shuffled the data.

Final dataset contains 84,865 tweets; 53,440 of them are labeled as normal, 31,425 of them are labeled as hateful. The ratio of the normal tweets in the dataset is 62,97% which indicates that our dataset is nearly balanced. You can see the balance from the Figure 1.

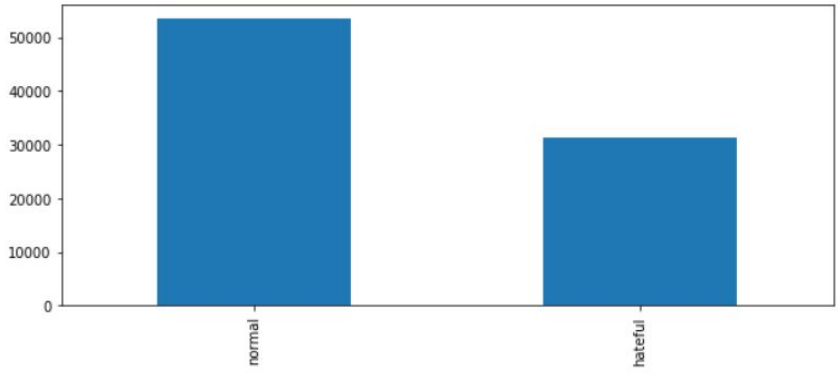


Figure 1-Tweets in the dataset

The second part of the preprocessing is feature extraction. Since we have text for the features, we need to have some number of transformational steps such as encoding categorical variables, feature scaling, and normalisation. To do these transformations, we have tried three methods: Pipeline, Word to Vector (Word2Vec), Bag of Words (BoW). Both Pipeline and Word2Vec is BoW based approaches with different methodologies.

3.1. Pipeline

Pipeline is a built in function in the preprocessing package of Scikit-Learn library and a tool to simplify the transformational step for feature extraction, mentioned above. What simply pipeline does is to vectorize, transform and classify; allows for a linear sequence of data transforms to be chained together culminating in a modeling process that can be evaluated. Formally, the purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters [6]. Moreover, pipeline method

allows us to use all the dataset without exceeding the RAM capacity and provides very fast training time.

Along with pipeline method, we have tried 8 algorithms such as Naive Bayes, Linear SVM, Logistic Regression, kNN, Decision Tree, Random Forest, Gradient Boosting, XGBoost. Since Random Forest, Gradient Boosting, and XGBoost are ensemble methods, higher performance was expected.

3.2. Word to Vector (Word2Vec)

Word2Vec transforms a text into a row of numbers. Word2Vec is a type of mapping that allows words with similar meaning to have similar vector representation. The idea behind is to use the surrounding words to represent the target words. The model we use is pre-trained by Google on a 100 billion word Google News corpus [7]. Also the common way to have similar vector representation is averaging the two word vectors. Therefore, we have followed the most common way.

Along with Word2Vec method, we have tried only Logistic Regression algorithm because the algorithm has a huge runtime due to significant amount of calculations between the vectors of each word in the dataset and the vectors in the pre-trained model.

3.3. Bag of Words (BoW)

BoW model is a way of representing text data when modeling text with machine learning algorithms. BoW involves two things: a vocabulary of known words, and a measure of the presence of known words. However, any information about the order or structure of words in the document is discarded. The model is only concerned with whether known words occur in the document, not where in the document. It creates a bag containing all the words in the dataset and vectorize for each input by labeling the words in the input tweet. BoW has seen great success in document classification problems, that is why we choose this model as well.

On the other hand, there is a trade-off between the feature numbers and runtime. As we increase the number of features for each input, runtime also significantly increases. Therefore, we put a limit that if a word occurs less than two times in the tweets we do not put the word into the features. This means before the limitation we had 10,713 features, after the limitation we have 6,371 features which is still big in terms of runtime and RAM capacity. The other problem is we have to limit the training and test set size due to the RAM capacity. Thus, for the total of training and test set size we have used 25,000, 20,000, and 10,000 tweets from

Along with BoW method, we have tried 6 algorithms such as Naive Bayes, SVM, Logistic Regression, kNN, Decision Tree, and RNN. Since RNN is a deep learning algorithm, higher performance was expected.

4. Results

4.1. Pipeline

In this method we have used Pipeline model and apply learning models accordingly. Dataset is splitted with the ratio 0.3. Total tweet count in training sample is 59,405, in test sample is 25,460. Normal tweet count in train set is 37,383, hateful tweet count in train set is 22,022. Normal tweet count in test set is 16,057, hateful tweet count in test set is 9,403. Distribution of the dataset into training and test set can be seen from the Figure 2,3.

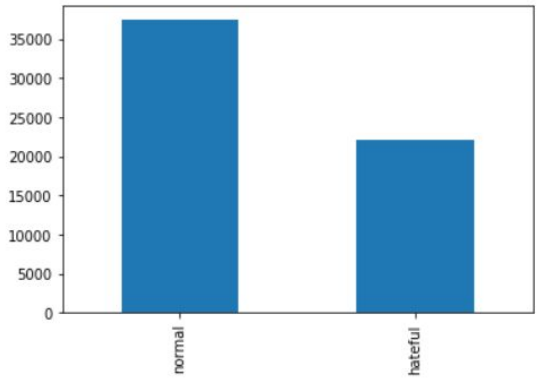


Figure 2-Tweets in the training set

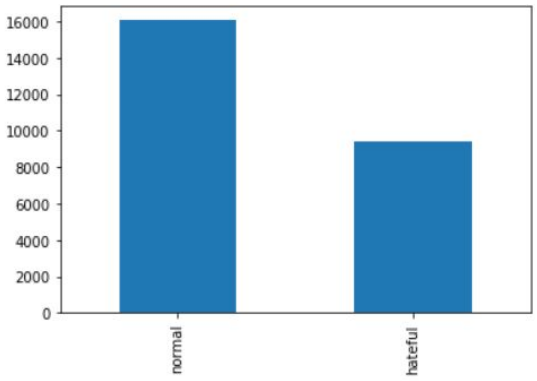


Figure 3-Tweets in the test set

4.1.1. Naive Bayes Classification

```
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

```
accuracy 0.8714454045561665
              precision    recall  f1-score   support

     hateful       0.88      0.76      0.81      9403
      normal       0.87      0.94      0.90     16057

   micro avg       0.87      0.87      0.87     25460
   macro avg       0.87      0.85      0.86     25460
weighted avg       0.87      0.87      0.87     25460

CPU times: user 718 ms, sys: 5.06 ms, total: 723 ms
Wall time: 729 ms
```

4.1.2. Linear Support Vector Machine (SVM)

```
SGDClassifier(loss='hinge', penalty='l2',alpha=1e-3,
              random_state=42, max_iter=5, tol=None)
```

```
accuracy 0.8553809897879026
              precision    recall  f1-score   support

     hateful       0.96      0.63      0.76      9403
      normal       0.82      0.99      0.90     16057

   micro avg       0.86      0.86      0.86     25460
   macro avg       0.89      0.81      0.83     25460
weighted avg       0.87      0.86      0.85     25460

CPU times: user 706 ms, sys: 2.82 ms, total: 708 ms
Wall time: 714 ms
```

4.1.3. Logistic Regression

```
LogisticRegression(n_jobs=1, C=1e5, random_state=None,
           solver='warn', tol=0.0001, verbose=0, warm_start=False)
```

```
accuracy 0.8515318146111548
              precision    recall  f1-score   support

     hateful       0.81      0.78      0.79      9403
      normal       0.87      0.90      0.88     16057

   micro avg       0.85      0.85      0.85     25460
   macro avg       0.84      0.84      0.84     25460
weighted avg       0.85      0.85      0.85     25460

CPU times: user 769 ms, sys: 3.91 ms, total: 773 ms
Wall time: 775 ms
```

4.1.4. K Nearest Neighbor (kNN)

```
KNeighborsClassifier(n_jobs=None, n_neighbors=3, p=2,
           weights='uniform')
```

```
accuracy 0.7993322859387274
              precision    recall  f1-score   support

     hateful       0.83      0.58      0.68      9403
      normal       0.79      0.93      0.85     16057

   micro avg       0.80      0.80      0.80     25460
   macro avg       0.81      0.75      0.77     25460
weighted avg       0.80      0.80      0.79     25460

CPU times: user 25.6 s, sys: 1.99 s, total: 27.6 s
Wall time: 27.6 s
```

4.1.5. Decision Tree Classification

```
DecisionTreeClassifier(min_weight_fraction_leaf=0.0, presort=False,
            random_state=None,splitter='best')
```

```
accuracy 0.872152395915161
              precision    recall  f1-score   support

     hateful       0.83      0.83      0.83      9403
      normal       0.90      0.90      0.90     16057

   micro avg       0.87      0.87      0.87     25460
   macro avg       0.86      0.86      0.86     25460
weighted avg       0.87      0.87      0.87     25460

CPU times: user 870 ms, sys: 11.9 ms, total: 882 ms
Wall time: 883 ms
```

4.1.6. Random Forest Classification

```
RandomForestClassifier(oob_score=False, random_state=None,
            verbose=0, warm_start=False)
```

```
accuracy 0.8960722702278083
              precision    recall  f1-score   support

     hateful       0.87      0.84      0.86      9403
      normal       0.91      0.93      0.92     16057

   micro avg       0.90      0.90      0.90     25460
   macro avg       0.89      0.88      0.89     25460
weighted avg       0.90      0.90      0.90     25460

CPU times: user 1.31 s, sys: 1.79 ms, total: 1.32 s
Wall time: 1.32 s
```

4.1.7. Gradient Boosting Classification

```
GradientBoostingClassifier(subsample=1.0, tol=0.0001,
            validation_fraction=0.1, verbose=0, warm_start=False)
```

```
accuracy 0.8732521602513748
              precision    recall  f1-score   support

     hateful       0.95      0.69      0.80      9403
      normal       0.84      0.98      0.91     16057

   micro avg       0.87      0.87      0.87     25460
   macro avg       0.90      0.84      0.85     25460
weighted avg       0.88      0.87      0.87     25460

CPU times: user 838 ms, sys: 899 µs, total: 839 ms
Wall time: 839 ms
```

4.1.8. XGBoost

```
XGBClassifier(reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
              seed=None, silent=True, subsample=1)
```

```
accuracy 0.8692851531814612
              precision    recall  f1-score   support

     hateful       0.96      0.68      0.79      9403
      normal       0.84      0.98      0.90     16057

   micro avg       0.87      0.87      0.87     25460
   macro avg       0.90      0.83      0.85     25460
weighted avg       0.88      0.87      0.86     25460

CPU times: user 917 ms, sys: 10.2 ms, total: 927 ms
Wall time: 936 ms
```

4.2. Word to Vector

4.2.1. Logistic Regression

First we load a word2vec model. It has been pre-trained by Google on a 100 billion word Google News corpus. We tokenize the text and apply the tokenization to “tweet” column, and apply word vector averaging to tokenized text. Then I build Logistic Regression model to see how logistic regression classifier perform on these word-averaging document features. It took 30 mins to run these steps. Comparing the results obtained by using Pipeline (85%) and BoW (86%), it can be said that Word2Vec is slightly better (+3% and +2%).

```
accuracy 0.8858208955223881
              precision    recall  f1-score   support

     hateful       0.89      0.79      0.84      9403
      normal       0.88      0.94      0.91     16057

   micro avg       0.89      0.89      0.89     25460
   macro avg       0.89      0.87      0.87     25460
weighted avg       0.89      0.89      0.88     25460
```

4.3. Bag of Words

We had not finalized our preprocessing during the implementation of this method. That's why we managed to use only some parts of the data due to memory problems. Regardless of the data size, number of hateful labeled tweets used is 4965 and the rest is tweets normal labeled.

4.3.1. Naive Bayes Classification

Data size: 20,000 Test size: 30% Accuracy: 70.4%		Precision	Recall	F1 Score
	Macro	64%	66%	64%
	Micro	70.4%	70.4%	70.4%
	Weighted	74%	70%	71%

Data size: 20,000 Test size: 25% Accuracy: 70.1%		Precision	Recall	F1 Score
	Macro	63%	66%	64%
	Micro	70.1%	70.1%	70.1%
	Weighted	74%	70%	71%

Data size: 20,000 Test size: 20% Accuracy: 69.6%		Precision	Recall	F1 Score
	Macro	63%	66%	64%
	Micro	69.6%	69.6%	69.6%
	Weighted	74%	69%	71%

Data size: 20,000 Test size: 10% Accuracy: 70.9%		Precision	Recall	F1 Score
	Macro	65%	68%	65%
	Micro	70%	70%	70%
	Weighted	74%	70%	72%

Data size: 25,000 Test size: 30% Accuracy: 72.8%		Precision	Recall	F1 Score
	Macro	62%	66%	63%
	Micro	72%	72%	72%
	Weighted	77%	72%	74%

4.3.2. Support Vector Machine (SVM)

Data size: 20,000 Test size: 30% Accuracy: 85.9%		Precision	Recall	F1 Score
	Macro	81.4%	79.8%	80.4%
	Micro	85.9%	85.9%	85.9%
	Weighted	88.4%	85.9%	86.8%

4.3.3. Logistic Regression

To model the Logistic Regression we used different solver types such as “sag”, “saga” and “liblinear”. These solvers are algorithms to solve an optimization problem. For small datasets, ‘liblinear’ is a good choice, whereas ‘sag’ and ‘saga’ are faster for large ones. For multiclass problems, only ‘newton-cg’, ‘sag’, ‘saga’ and ‘lbfgs’ handle multinomial loss; ‘liblinear’ is limited to one-versus-rest schemes [8]. Also, different tolerance values are applied to see the effects on results. The default tolerance value is 10^-4 and we used to model with tolerance value 10^-2 and 10^0 to.

4.3.3.1. Model with “Sag” solver and (10^-4) default tolerance

Data size: 20,000 Test size: 30%		Precision	Recall	F1 Score

Accuracy: 85.90%	Macro	84.49%	76.37%	79.16%
	Micro	85.90%	85.90%	85.90%
	Weighted	85.55%	85.90%	84.96%

Test Size	30%	25%	20%
Accuracy	85.90%	86.28%	86.25%

4.3.3.2. Model with “Saga” solver and (10^-4) default tolerance

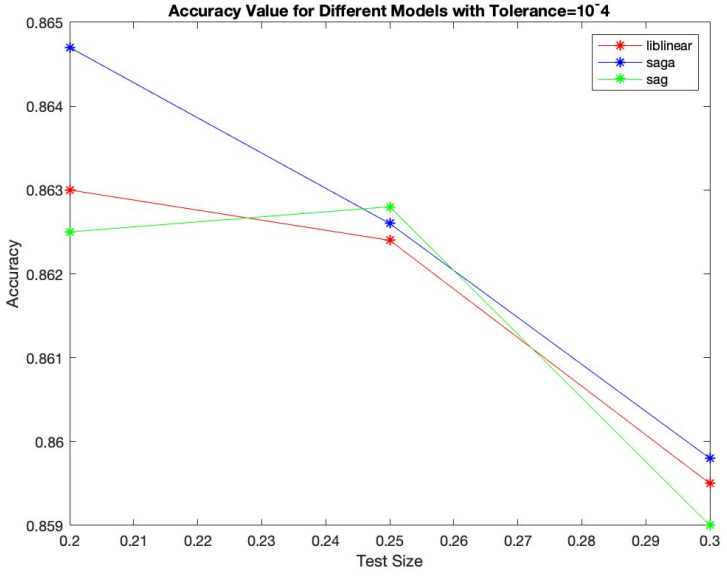
Data size: 20,000 Test size: 30% Accuracy: 85.98%		Precision	Recall	F1 Score
	Macro	84.81%	76.32%	79.19%
	Micro	85.98%	85.98%	85.98%
	Weighted	85.69%	85.98%	85.01%

Test Size	30%	25%	20%
Accuracy	85.98%	86.26%	86.47%

4.3.3.3. Model with “Liblinear” default solver and (10^-4) default tolerance

Data size: 20,000 Test size: 30% Accuracy: 85.95%		Precision	Recall	F1 Score
	Macro	84.54%	76.47%	79.25%
	Micro	85.95%	85.95%	85.95%
	Weighted	85.61%	85.95%	85.02%

Test Size	30%	25%	20%
Accuracy	85.95%	86.24%	86.30%



4.3.3.4. Model with “Sag” solver and (10^-2) default tolerance

Data size: 20,000 Test size: 30% Accuracy: 86.11%		Precision	Recall	F1 Score
	Macro	85.38%	76.21%	79.24%
	Micro	86.11%	86.11%	86.11%
	Weighted	85.92%	86.11%	85.09%

Test Size	30%	25%	20%
Accuracy	86.11%	86.38%	86.50%

4.3.3.5. Model with “Saga” solver and (10^-2) default tolerance

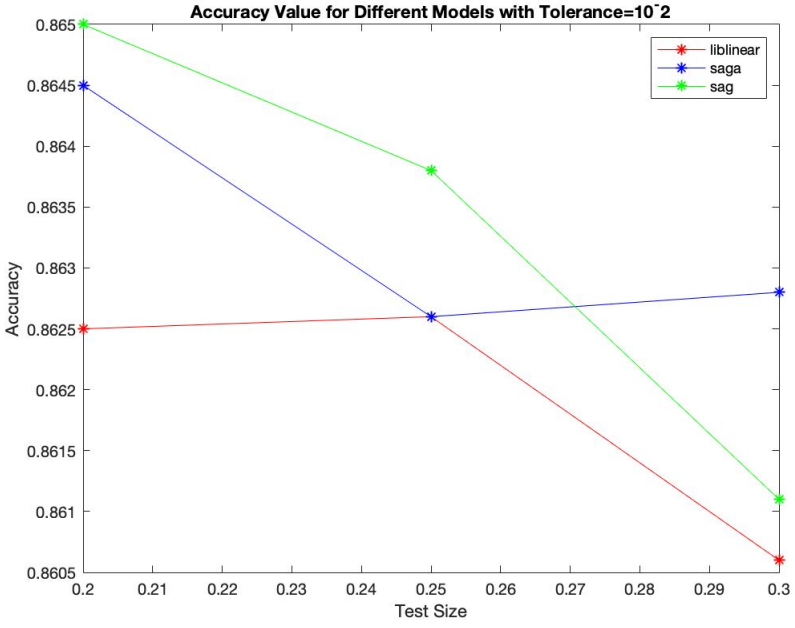
Data size: 20,000 Test size: 30% Accuracy: 86.28%		Precision	Recall	F1 Score
	Macro	84.03%	76.15%	79.33%
	Micro	86.28%	86.28%	86.28%
	Weighted	86.21%	86.28%	85.20%

Test Size	30%	25%	20%
Accuracy	86.28%	86.26%	86.45%

4.3.3.6. Model with “Liblinear” default solver and (10^-2) tolerance

Data size: 20,000 Test size: 30% Accuracy: 86.06%		Precision	Recall	F1 Score
	Macro	84.66%	76.69%	79.46%
	Micro	86.06%	86.06%	86.06%
	Weighted	85.73%	86.06%	85.17%

Test Size	30%	25%	20%
Accuracy	86.06%	86.26%	86.25%



4.3.3.7. Model with “Sag” solver and (10^0) default tolerance

Data size: 20,000 Test size: 30% Accuracy: 83.71%		Precision	Recall	F1 Score
	Macro	85.81%	69.83%	73.25%
	Micro	83.71%	83.71%	83.71%
	Weighted	84.46%	83.71%	81.44%

Test Size	30%	25%	20%
Accuracy	83.71%	84.32%	84.77%

4.3.3.8. Model with “Saga” solver and (10^0) default tolerance

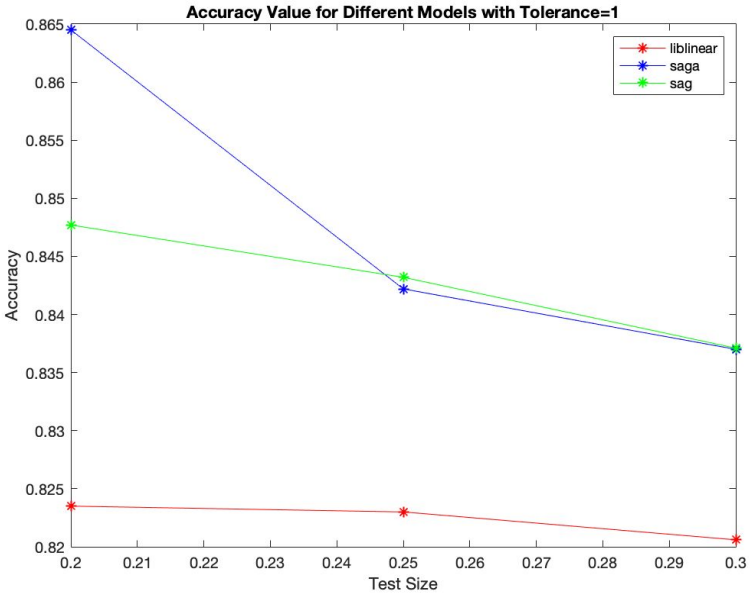
Data size: 20,000 Test size: 30% Accuracy: 83.70%		Precision	Recall	F1 Score
	Macro	85.98%	69.71%	73.14%
	Micro	83.70%	83.70%	83.70%
	Weighted	84.52%	83.70%	81.39%

Test Size	30%	25%	20%
Accuracy	83.70%	84.22%	86.45%

4.3.3.9. Model with “Liblinear” solver and (10^0) default tolerance

Data size: 20,000 Test size: 30% Accuracy: 82.06%		Precision	Recall	F1 Score
	Macro	84.68%	66.38%	69.26%
	Micro	82.06%	82.06%	82.06%
	Weighted	83.08%	82.06%	78.98%

Test Size	30%	25%	20%
Accuracy	82.06%	82.30%	82.35%



4.3.4. K Nearest Neighbor (kNN)

KNN algorithm uses neighbors of each point to predict its label. For example, if the number of nearest neighbors under consideration is equal to 1, then the label of the observation will be same with the nearest neighbor. However, this one neighbor could be noise which could result in wrong classification. Therefore, increasing the number of neighbors should increase the accuracy of the model since the impact of noise is reduced and the model will become more generalizable. But there will be an optimum value for number of nearest neighbors.

The main objective of KNN algorithm is to observe the change in confusion matrix and the accuracy with altering number of nearest neighbors. Therefore, number of training and test samples are fixed. Only number of nearest neighbors is changed.

With n = 5:
Accuracy = 0.8245

Confusion Matrix for N=5			
	Precision	Recall	F1-score
Hateful	0.71	0.53	0.61
Normal	0.85	0.93	0.89
Micro Avg.	0.82	0.82	0.82
Macro Avg.	0.78	0.73	0.75
Weighted Avg.	0.82	0.82	0.82

With n = 10:
Accuracy = 0.84083

Confusion Matrix for N=10			
	Precision	Recall	F1-score
Hateful	0.80	0.50	0.61
Normal	0.85	0.96	0.90
Micro Avg.	0.84	0.84	0.84
Macro Avg.	0.83	0.73	0.76
Weighted Avg.	0.84	0.84	0.83

With n = 15:
Accuracy = 0.826666

Confusion Matrix for N=15			
	Precision	Recall	F1-score
Hateful	0.89	0.37	0.52
Normal	0.82	0.98	0.89
Micro Avg.	0.83	0.83	0.83
Macro Avg.	0.85	0.68	0.71
Weighted Avg.	0.84	0.83	0.80

With n=20:
Accuracy = 0.81783

Confusion Matrix for N=15			
	Precision	Recall	F1-score
Hateful	0.90	0.32	0.47
Normal	0.81	0.99	0.89
Micro Avg.	0.82	0.82	0.82
Macro Avg.	0.86	0.65	0.68
Weighted Avg.	0.83	0.82	0.78

From the results, it is seen that the maximum accuracy is achieved at n=10. The lowest one is the n=20. Which means that increasing number of nearest neighbors does not increase the accuracy value but there is an optimum value. This optimum number of nearest neighbors could be found by using methods like cross-validation.

4.3.5. Decision Tree

A decision tree classifier is built the given parameters. According to criterion, changing the min_samples_leaf and max_depth parameters, accuracy values are obtained which are showed in below tables.

```
DecisionTreeClassifier(criterion=' ', max_depth= ,
            min_samples_leaf=, min_samples_split=2,
            random_state=100, splitter='best')
```

min_samples_leaf	criterion = “entropy”	criterion = “gini”
1	0.8598333333333333	0.8663629460895975
5	0.86	0.8656036446469249
10	0.8596666666666667	0.8673753480131612
50	0.8583333333333333	0.8661098456087066
100	0.8511666666666666	0.8613009364717793
250	0.8336666666666667	0.8263730701088332
500	0.7768333333333334	0.7853707922045052

max_depth	criterion = “entropy”	criterion = “gini”
3	0.8230827638572513	0.8230827638572513
5	0.846	0.84763351050367
7	0.8555	0.8595292331055429
9	0.86	0.8656036446469249
11	0.8638333333333333	0.8663629460895975
13	0.8665	0.8691470513793976
15	0.869	0.8704125537838522
20	0.8743333333333333	0.8613009364717793
30	0.8748333333333334	0.8613009364717793
40	0.8781666666666667	0.8613009364717793

50	0.8788333333333334	0.8613009364717793
100	0.8763333333333333	0.8613009364717793

4.3.6. Recurrent Neural Network (RNN)

We have built the model using Keras library from Tensorflow. The model has 5 layers:

1. dense_1: Dense(512, input_shape=(max_words,))
2. activation_1: Activation('relu')
3. dropout_1: Dropout(0.5)
4. dense_2: Dense(num_classes)
5. activation_2: Activation('softmax')

Also, the model has a compiler with these parameters: loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'].

After creating the model data is fitted according to the given batch size, number of epochs, and validation_split = 0.1. Test size is 30% of the dataset.

Batch size: 128
Epochs: 20
Test Accuracy: 89.17%

```
Train on 53464 samples, validate on 5941 samples
Epoch 1/20
53464/53464 [==============================] - 8s 149us/step - loss: 0.3348 - acc: 0.8741 - val_loss: 0.2856 - val_acc: 0.8975

Epoch 20/20
53464/53464 [==============================] - 8s 154us/step - loss: 0.0910 - acc: 0.9672 - val_loss: 0.4215 - val_acc: 0.8946
```

Batch size: 256
Epochs: 20
Test Accuracy: 89.33%

```
Train on 53464 samples, validate on 5941 samples
Epoch 1/20
53464/53464 [==============================] - 5s 100us/step - loss: 0.0860 - acc: 0.9701 - val_loss: 0.4251 - val_acc: 0.8985

Epoch 20/20
53464/53464 [==============================] - 5s 98us/step - loss: 0.0725 - acc: 0.9740 - val_loss: 0.5025 - val_acc: 0.8973
```

Batch size: 512
Epochs: 20
Test Accuracy: 89.34%

```
Train on 53464 samples, validate on 5941 samples
Epoch 1/20
53464/53464 [==============================] - 4s 76us/step - loss: 0.0699 -
acc: 0.9748 - val_loss: 0.5036 - val_acc: 0.8968

Epoch 20/20
53464/53464 [==============================] - 4s 76us/step - loss: 0.0642 - acc: 0.9764 -
val_loss: 0.5414 - val_acc: 0.8975
```

Batch size: 1024
Epochs: 20
Test Accuracy: 89.33%

```
Train on 53464 samples, validate on 5941 samples
Epoch 1/20
53464/53464 [==============================] - 3s 59us/step - loss: 0.0630 -
acc: 0.9765 - val_loss: 0.5408 - val_acc: 0.8970

Epoch 20/20
53464/53464 [==============================] - 4s 68us/step - loss: 0.0602 - acc: 0.9772 -
val_loss: 0.5598 - val_acc: 0.8961
```

Batch size: 512
Epochs: 10
Test Accuracy: 89.17%

```
Train on 53464 samples, validate on 5941 samples
Epoch 1/10
53464/53464 [==============================] - 4s 75us/step - loss: 0.0617 -
acc: 0.9767 - val_loss: 0.5665 - val_acc: 0.8972

Epoch 10/10
53464/53464 [==============================] - 4s 75us/step - loss: 0.0616 - acc: 0.9766 -
val_loss: 0.5676 - val_acc: 0.8951
```

Batch size: 512
Epochs: 30
Test Accuracy: 89.29%

```
Train on 53464 samples, validate on 5941 samples
Epoch 1/30
53464/53464 [==============================] - 4s 75us/step - loss: 0.0624 -
acc: 0.9769 - val_loss: 0.5704 - val_acc: 0.8953

Epoch 30/30
53464/53464 [==============================] - 4s 76us/step - loss: 0.0588 - acc: 0.9778 -
val_loss: 0.6230 - val_acc: 0.8978
```

5. Discussion

Best accuracies received from each algorithm according to preprocessing model is given below;

	Pipeline	Word2Vec	BoW
Naive Bayes	87.14%	-	72.8%
SVM	85.54%	-	85.9%
Logistic Regression	85.15%	88.58%	87%
KNN	79.93%	-	84.10%
Decision Tree	87.22%	-	87.88%
Random Forest	89.61%	-	-
Gradient Boosting Classification	87.33%	-	-
XGBoost	86.93%	-	-
RNN	-	-	89%

We improved the quality of preprocessing while using pipeline method for algorithms.

We can see that the accuracy of Naive Bayes algorithm increases when we use pipeline instead of Bag of Words. This can be the result of both the improvement in preprocessing or the increase in the quality and the amount of data we use.

In the KNN algorithm, we saw that there is no linear relation between accuracy and the number of nearest neighbors but there is an optimal “n” value which could be found by applying cross-validation. Additionally, as the number of neighbors increases, the computational time also increases because model has to consider the desired number of neighbors for all points.

In Logistic Regression model, we saw that different types of solvers could end up with different values of accuracy. Although these variations are small, one could decide which solver is best for the existing dataset since each solver has advantages and

disadvantages over another for dataset sizes. Also preprocessing model also affects the efficiency of the model. Logistic Regression works best with the Word2Vec model.

Runtime of SVM model is really high when it tries to fit data to model. Thus we trained and tested one fifth of our dataset. Reason of its slowness is that it has two primary parameters thus it takes N^2 to train data. Therefore even though it has a high accuracy, SVM is not an effective model for large datasets.

Accuracy of RNN changes according to the given epoch numbers and batch sizes. The optimal batch size, in our case, seems to be 512 whereas the optimal number epochs is 20.

Among the algorithms applied with Pipeline, Random Forest has the highest accuracy and the rest of them also work fine; however, kNN has the lowest accuracy.

6. Conclusions

The most problematic part of applying models to the dataset is the computational time for the bag of words method. While Naive Bayes, Logistic Regression and KNN algorithms work relatively faster, SVM and RNN are slower. This is why we get computational error in Google Colab. To overcome this problem, we had to remove some of the data and decrease the number of features to decrease the computational time. However, we did not face such a problem while applying pipeline.

In conclusion, we obtain different results with the same algorithm when a different model is applied. Furthermore, among the preprocessing models, Pipeline is faster than others in terms of runtime and easy to use with a lot algorithms. Although, Word2Vec and BoW are also suitable for this kind of problem, runtime of Word2Vec is significantly slow and BoW has limitations. Finally, RNN and Random Forest have given the best accuracy. So, they might be the most suitable algorithms for tweet classification problem.

7. Appendix

All members contributed to the report, extracted the results of the algorithms implemented and commented on them.

Nurefşan Müsevitoğlu

- Improved preprocessing. Implemented all algorithms used with pipeline; Logistic Regression with Word2Vec and RNN.

Yasin Balcancı

- Worked on implementing BoW Gaussian Naive Bayes and Logistic Regression, contributed to SVM.

Semih Teker

- Worked on implementing BoW Gaussian Naive Bayes and Logistic Regression.

Balkır Göka

- Worked on implementing BoW Gaussian Naive Bayes and KNN.

Muammer Tan

- Preprocessed the dataset, worked on implementing BoW SVM.

References

[1] Large Scale Crowdsourcing and Characterization of Twitter Abusive Behavior. [Online] Available: https://arxiv.org/pdf/1802.00393.pdf Accessed: March 15, 2019.

[2] Number of social media users worldwide from 2010 to 2021 (in billions). [Online] Available: https://www.statista.com/statistics/278414/number-of-worldwide-social-network-users/

[3] Number of monthly active Twitter users worldwide from 1st quarter 2010 to 1st quarter 2019 (in millions). [Online] Available:https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/

[4] How should social media platforms combat misinformation and hate speech?. [Online]. Available: https://www.brookings.edu/blog/techtank/2019/04/09/how-should-social-media-platforms-combat-misinformation-and-hate-speech/

[5] EU says Facebook, Google and Twitter are getting faster at removing hate speech online. [Online]. Available: https://www.cnbc.com/2019/02/04/facebook-google-and-twitter-are-getting-faster-at-removing-hate-speech-online-eu-finds--.html

[6] sklearn.pipeline.Pipeline. [Online] Available: https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html Accessed: May 12, 2019.

[7] GoogleNews-vectors-negative300.bin.gz. [Online] Available: https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit Accessed: May 12, 2019.

[8] Scikit-learn.org. (2019). sklearn.linear_model.LogisticRegression — scikit-learn 0.21.0 documentation. [Online] Available at: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html [Accessed 12 May 2019].