**CS583 – Research Project**

**Abstract**

Sentiment analysis is the process of analyzing digital text to identify if the message's emotional tone is neutral, positive, or negative. With the prevalence of social media, Sentiment Analysis as a field has been brought to the forefront as it is now used for various applications such as product marketing, political analysis, customer support and feedback, market research and so on.

In this project, I aim to build a model that can classify the sentiment of tweets into one of three classes: positive, negative or neutral. As a part of the set up for this project, we are provided with two datasets: a collection of tweets related to Barack Obama, and the other a collection of tweets related to Mitt Romney. These tweets were downloaded using the Twitter API during the height of the Presidential race in the United States of America in the year 2012.

Using this data, I compare different methods of feature-extraction, models for classification, and evaluate their performance. I also discuss what I’ve learned from undertaking this process, and what could be some improvements to make in the future.

**Introduction**

Sentiment analysis is a widely used in the industry today to find what customers have to say about brands, political entities, and so on.

**Techniques**

To start with the training of the models, I load the tweets for Barack Obama from the excel sheet into a dataframe in the obama.ipynb notebook. The data that we are presented with requires some cleaning to get into a format that would work for us, and for that I first drop the first row of the dataframe since that has data that we do not need.

The next step that I perform is to drop the columns 'Unnamed: 0', 'date', 'time', 'Unnamed: 5'. ‘Unnamed: 0’ and ‘Unnamed: 5’ are an empty column with all nan values. ‘date’ and ‘time’ are two other columns that we do not use in the process, so I drop those too.

I also rename the columns ‘Unnamed: 4’ to ‘class’ and ‘Anootated tweet’ to ‘tweet’, to make things easier to work with and understand further into the process.

When checking the values present in the column ‘class’, we see repeated entries for the values. This is because some of the values are strings, while the others are integers. To keep only the -1, 0 and 1 columns, I drop everything from the dataframe that is not one of these in integer or string data types, and then convert the entire column to integers for consistency. We can check the value\_counts of the ‘class’ column again after these steps, and we see that all the data is in integer format, just as we need.

The last part is performing a quick analysis and printing the distribution of the classes among our data as a bar chart.

These steps are repeated for the tweets concerning Mitt Romney in the romney.ipynb notebook.

The next steps that I perform are cleaning the tweets themselves, and tokenizing the strings. There are two functions that are responsible for this – clean() and tokenize(). In the function clean(), I remove all the strings from the tweets that are not needed – such as hashtags, accounts referred to with ‘@’ handles, hyperlinks and so on. Tokenize() uses English stopwords from the nltk library to remove commonly used English language words that do not carry any meaning – words such as ‘a’, ‘the’, ‘is’, ‘and’ and so on. The words are then tokenized using RegexpTokenizer from nltk.tokenize, and the tokenizedstrings are returned. These functions are called for every tweet in the dataframe using a simple lambda function.

The next step is to remove all words that have a length of less than 2. Then, we create a frequency distribution of all the words in all the tweets using FreqDist from nltk.probability. This frequency distribution is then used to remove words which have a frequency of less than 2 in the distribution from the tweets.

Now, we are ready to lemmatize our tweets. Lemmatization is the process of reducing words to their base or root form, which helps improve accuracy. To perform this process, we use a function called lemmatizer(), which takes a string as an input and returns a lemmatized version of the text. The input is tokenized into words using nltk.word\_tokenize, after which we tag each token with the part of speech it is related to using nltk.pos\_tag. These tags are then converted into a wordnet format using a function pos\_tagger(). Pos\_tagger() converts parts-of-speech tags from the NLTK format to a format compatible with the WordNet lemmatizer. Once this function returns its data, we add each word to a list. If the tag is None, indicating that the word could not be tagged, it is appended as is. If the word has a tag, it is lemmatized using WordNetLemmatizer.lemmatize along with its tag, and then appended to the list. The lemmatized sentence is then returned as a string. This function is applied to all tweets using a lambda function.

At this point, we can drop all the extra columns that we have created as a part of the steps above. I drop the columns ‘tweet\_token’, ‘tweet\_string’ and ‘tweet\_string\_fdist’ since we do not need them anymore. I also drop all rows that are empty using dropna, to ensure that our dataset is as clean as possible.

The last step is to create a train-test-split of our dataset, so that we can calculate the performance of our models on some sample data.

To create models for our data, we first need to vectorize our tweets. I explore using 2 methods of vectorization – TF-IDF from sklearn and Word2Vec from genism. TfidfVectorizer is used by simply calling fit\_transform() on the train split, and transform() on the test split. For the Word2Vec vectorization we first convert our train and test sets into tokens, and then use the MeanEmbeddingVectorizer class to convert a list of words into word vectors using a trained Word2Vec model. The model is trained on the tokenized tweets, and then we create a dictionary where the keys are words and values are the word vectors. This is used to create an instance of the class, and then we fit the training data and transform it and the test data into vectors. To ensure that we do not get any negative values in our vectors, we use a MinMaxScaler() to move our data range into the positive space. Negative numbers in the vectors causes errors when creating and using the models, and thus this step is needed to avoid those.

Now that we have our vectors ready, we can start applying our models. We first take a look at supervised learning algorithms, where we use Logistic Regression, Naïve Bayes, SVM and KNN from sklearn on the TF-IDF vectors and Word2Vec vectors.

For both TF-IDF and Word2Vec, we follow similar steps when training and predicting.

First, we create a Logistic Regression model using sklearn’s library. Through experimentation, I settled on using the ‘saga’ solver, C = 5 and penalty = l2. We train on the train dataset vectors created above, and predict on the test dataset vectors. Finally, the performance metrics are written to a dataframe for easy analysis.

The second model we use is Naïve Bayes from sklearn, and follow similar steps as above.

We then use SVM with kernel = ‘linear’, and fit and predict as before.

The last model that we use is KNN. Using GridSearchCV, we find the optimal value for the number of neighbors in a range from 1 to 25, and then fit and predict as above.

After all these models have been run, we compare the performance of each of these models, and to make things easier, we use a Voting Classifier from sklearn to make the decision for us. Voting Classifier is an ensemble method, which trains using numerous models, and predicts an output class based on majority voting from each of these models. I use soft voting, where the output class is predicted based on the average probability given to a class. To do this, I create a voting classifier with Logistic Regression, Naïve Bayes and SVM, fit on the training vectors and then predict on the test vectors.

This achieves a decent accuracy, but we can do better. At the start of this project, I drop all records that are not in class -1,0 and 1. This leaves out 1544 records that are not in any of these classes and belong to class 2. Class 2 is defined as tweets having both positive and negative sentiment. Thus, we can treat this data as unlabeled and perform Semi-Supervised learning.

To do so, I take the tweets from the dataset that have class = 2, and apply the same data preprocessing steps from above. Then, we predict the classes for these records, and merge these tweets with their predicted labels to the original dataframe. We take this dataframe and perform our train-test-split, convert the train and test datasets into TF-IDF vectors and perform a prediction again. This is the model that we use for our final classification on the sample data provided.

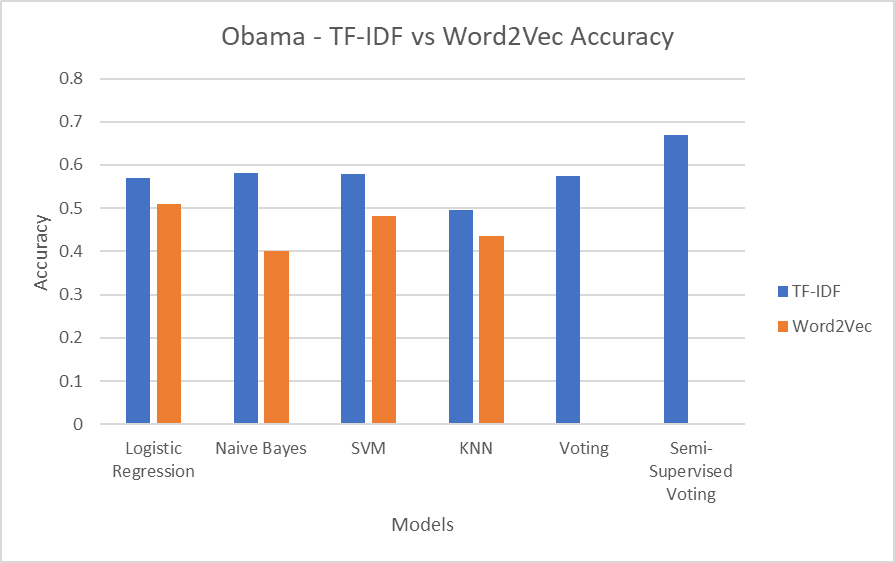
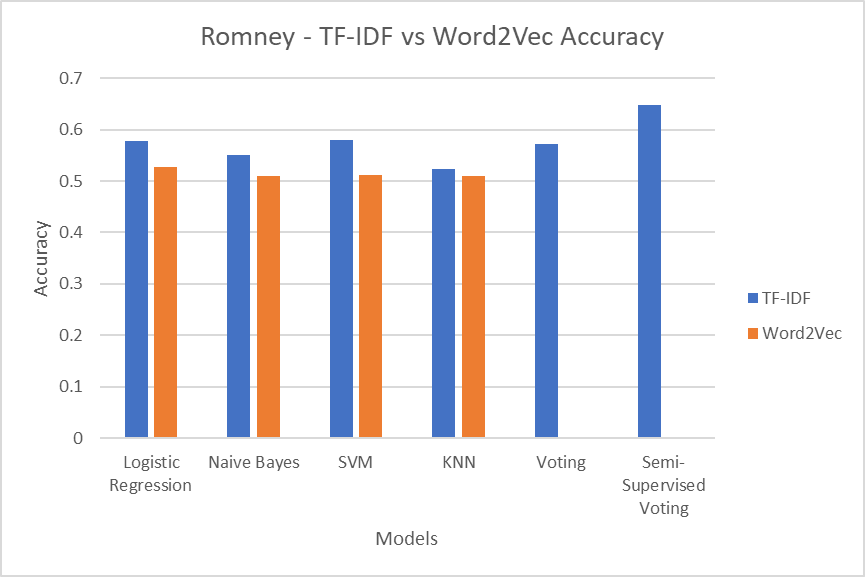
**Results**

**Obama:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model** | **Vectorization** | **Accuracy** | **Precision** | **Recall** | **F1 Score** |
| Logistic Regression | TF-IDF | 0.5698 | 0.5726 | 0.5698 | 0.5706 |
| Logistic Regression | Word2Vec | 0.5102 | 0.5179 | 0.5102 | 0.5114 |
| Naive Bayes | TF-IDF | 0.5804 | 0.588 | 0.5804 | 0.5812 |
| Naive Bayes | Word2Vec | 0.4009 | 0.3924 | 0.4009 | 0.393 |
| SVM | TF-IDF | 0.5787 | 0.581 | 0.5787 | 0.5794 |
| SVM | Word2Vec | 0.4818 | 0.5026 | 0.4818 | 0.4816 |
| KNN | TF-IDF | 0.4951 | 0.5432 | 0.4951 | 0.4821 |
| KNN | Word2Vec | 0.4356 | 0.4345 | 0.4356 | 0.4262 |
| Voting | TF-IDF | 0.5742 | 0.5765 | 0.5742 | 0.5748 |
| Semi- Supervised Voting | TF-IDF | 0.6695 | 0.6716 | 0.6695 | 0.6685 |

**Romney:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model** | **Vectorization** | **Accuracy** | **Precision** | **Recall** | **F1 Score** |
| Logistic Regression | TF-IDF | 0.5779 | 0.5617 | 0.5779 | 0.5605 |
| Logistic Regression | Word2Vec | 0.5283 | 0.511 | 0.5283 | 0.4356 |
| Naive Bayes | TF-IDF | 0.5504 | 0.5952 | 0.5504 | 0.4491 |
| Naive Bayes | Word2Vec | 0.5106 | 0.3154 | 0.5106 | 0.3515 |
| SVM | TF-IDF | 0.5796 | 0.5612 | 0.5796 | 0.5551 |
| SVM | Word2Vec | 0.5124 | 0.2625 | 0.5124 | 0.3472 |
| KNN | TF-IDF | 0.523 | 0.4843 | 0.523 | 0.4703 |
| KNN | Word2Vec | 0.5106 | 0.4695 | 0.5106 | 0.4069 |
| Voting | TF-IDF | 0.5726 | 0.5553 | 0.5726 | 0.5293 |
| Semi- Supervised Voting | TF-IDF | 0.6488 | 0.6465 | 0.6488 | 0.6309 |

The above two charts display the accuracy for all the models implemented for the Obama and Romney datasets. With just Supervised learning, I was able to achieve an accuracy of 58% on the Obama dataset and 57.7% on the Romney dataset. This could be increased by 1% if in the data preprocessing stage, I removed all words with length of 1 instead of 2 as it is right now, but that decreased the accuracy for the final model by 2% which is a tradeoff that I did not want to make. We can also see from the above charts that Word2Vec lags in accuracy for all models, and sometimes significantly. TF-IDF is clearly the superior vectorization method, and it is the one used for the final model.

Of the models that I tried to use, Logistic Regression, Naïve Bayes and SVM performed the best, while KNN was the worst in both datasets. As expected, Voting Classifier performed as well as the individual models, and when we introduced additional training data in the Semi-Supervised learning Voting Classifier, the accuracy shot up to 67% for the Obama dataset and 65% for the Romney dataset.

**Conclusion**

In conclusion, the results that we found were interesting. From what I experienced during this research project, for supervised learning, there is no single model that is the best. Logistic Regression, Naïve Bayes and SVM all perform equally well, which is why I chose the approach of implementing them in a Voting Classifier.

**Lessons Learned**