Machine Learning Engineer Nanodegree

Capstone Project

Iuri Barbosa
December 09th, 2016

I. Definition

Project Overview

Sentiment Analysis is a major area in Natural Language Processing and consists of extracting underlying sentiments from texts. Its ultimate goal is to determine whether the sentiment related to the text is positive, negative or neutral [1].

A great source of information for sentiment analysis is found in reviews from some review websites, such as *Amazon, TripAdvisor,* and *Glassdoor*, to list a few. In the movie and series industry, we have *IMDB (The Internet Movie Database)* and *Rotten Tomatoes*, for example. For this project, it will be analyzed the latter group, specifically IMDB reviews for top crime series [7].

Problem Statement

The ultimate goal for this project is to assess the performance of multiple supervised models in classifying IMDB reviews of top crime series in positive or negative sentiments. For this project, the default number of series analyzed is the **top 25**.

To achieve this goal, the **target class** will be defined as the **rating** that comes along with the **review text**, both given by the user in a review. In this way, the ratings will represent the underlying sentiment associated with review texts. As ratings vary from 1 to 10, two different ranges of ratings will express negative and positive sentiments.

Once there is a great number of movies and series reviews in IMDB, the efforts of this project will be on crime related series. There isn't a special reason to have selected this topic, and the results and analysis are reproducible for other topics and subjects. It is worth considering that limiting the analysis within a

single context will be useful for creating a shared and specialized **vocabulary** for the set of review texts.

The projected can be broken down into five major tasks, which will be discussed in the next sections, but can be summarized as:

1. Download the dataset made up of series reviews (review texts and ratings) using a crawler.
2. Analyze and visualize the downloaded data in order to have some insights about it.
3. Preprocess review texts before fitting into supervised models and explore different customization options.
4. Train and test a set of classifiers as well as tuning their parameters and assessing their performance.
5. Discuss the results achieved by the models and draw a conclusion about the overall project.

An **optimal solution** for this project would represent the optimal classifier along with its optimized parameters in a number of scenarios. For this reason, model's performance will be compared to the benchmark model (defined in the next section).

Metrics

It is worth mentioning that the downloaded data might change depending on which time it was downloaded since reviews pages for the series are dynamic. The results discussed in this project are related to the reviews downloaded on **1st December, 2016**.

In the next section, it will be shown that the downloaded dataset is unbalanced. Approximately 78% of the downloaded reviews are classified as having a positive sentiment. For this reason, a simple **score function**, such as **accuracy**, would **not** represent how precise is the model's performance [2]. An optimal metric would take into account false negatives and false positives.

Imagine, for example, a hypothetical situation where 99% of a dataset has a positive label, and the remaining 1% has a negative label. On top of that, we have a model that always outputs positive labels. Although, this model has a high accuracy (99%), it is clearly not the intended solution.

In those situations, an excellent alternative to evaluate model performance is the **F1 score** [3]. F1 score stands out among other metrics because it combines **recall** and **precision** harmonically. The formulas for these metrics are represented below:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

where TP = true positive, FP = false positive, TN = true negative and FN = false negative.

For having this property of balancing recall and precision in a single metric, F1 score is the metric that will be used for assessing model's performance.

II. Analysis

Data Exploration and Exploratory Visualization

Web Crawler

One of the hardest tasks in Data Science and Machine Learning is to collect reliable data to perform some analysis. In a number of situations, the data must be collected from unstructured sources, such as websites.

Web crawler is the name of a system which browses web pages in automated manner, and it can be used to retrieve contents from these unstructured sources of information [4].

In general, a regular web crawler would follow these three steps:

1. Request web page content and store it temporarily.
2. Parse the content into a navigable HTML tree.
3. Search, extract and store the useful information found in the tree.

In this project, the *requests* library [5] is used to perform step 1; steps 2 and 3 can be done using another python library called, *Beautiful Soup* [6].

Being more specific about the step 3, we can subdivided it as follows:

1. Extract a list of IMDB series identifiers (series IDs) from the home page of top crime series. **Series ID** represents a unique idenfier for series in the IMDB system.
2. For each series IDs, access series reviews page, extract and download the **reviews** related to the series. The most important reviews features are the **review text** (written by the user to describe the series) and **review rating** (associated with either positive or negative sentiment).

In the end, the downloaded data is organized in a single parent directory, and each series has its own directory. Within each series directory, there are directories representing each rating number form 1 to 10. Finally, each rating directory has the review texts as text files (.txt). A preview of this folder structured is shown in the picture below:

Illustration 1: Dataset structure for the top 25 crime series in IMDB.

Rating Ranges

It is necessary to define which ratings represent positive and negative sentiments. For example, it can be defined that ratings below or equivalent to 5 stars represent review texts with a negative sentiment, while ratings above 5 stars have positive sentiment. Even better, we can be more specific and state that ratings below or equivalent to 3 stars represent negative sentiment and that ratings above or equivalent to 8 starts positive sentiment.

For this discussion, we will take into account **review texts** with negative sentiment as having ratings within the range *[1, 3]* and positive sentiment within *[8, 10]* (inclusive). These intervals were chosen because they tend to represent stronger opinions, either positive or negative, while the interval in the middle ([4, 7] stars) might represent mixed opinions. This way, we can eliminate a great level of noise in the dataset.

Class and Rating Distributions

With these definitions in mind, by exploring the class distribution (negative and positive sentiments) and the rating distribution, we have:

1. There are 4253 reviews in total, from which 3303 are positive (almost 78%) and 950 negative.
2. Most of the review ratings are 10 stars, which is something expected once the reviews come from the top 25 crime series.

The class and rating distributions can be visualized by the next two images:

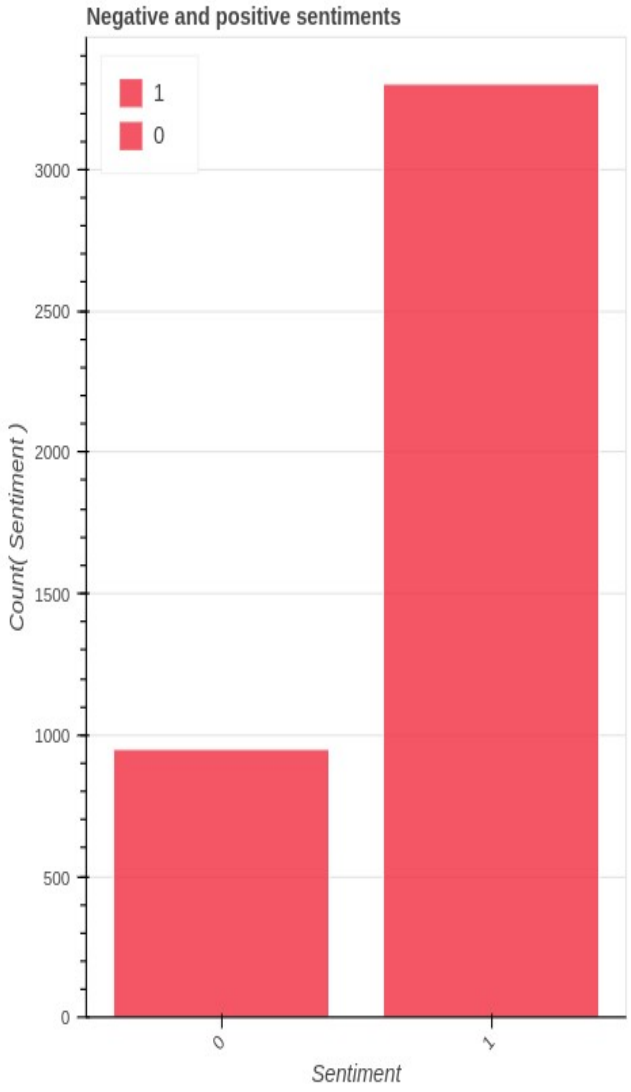

Illustration 2: Sentiment distribution (0 negative, 1 positive)

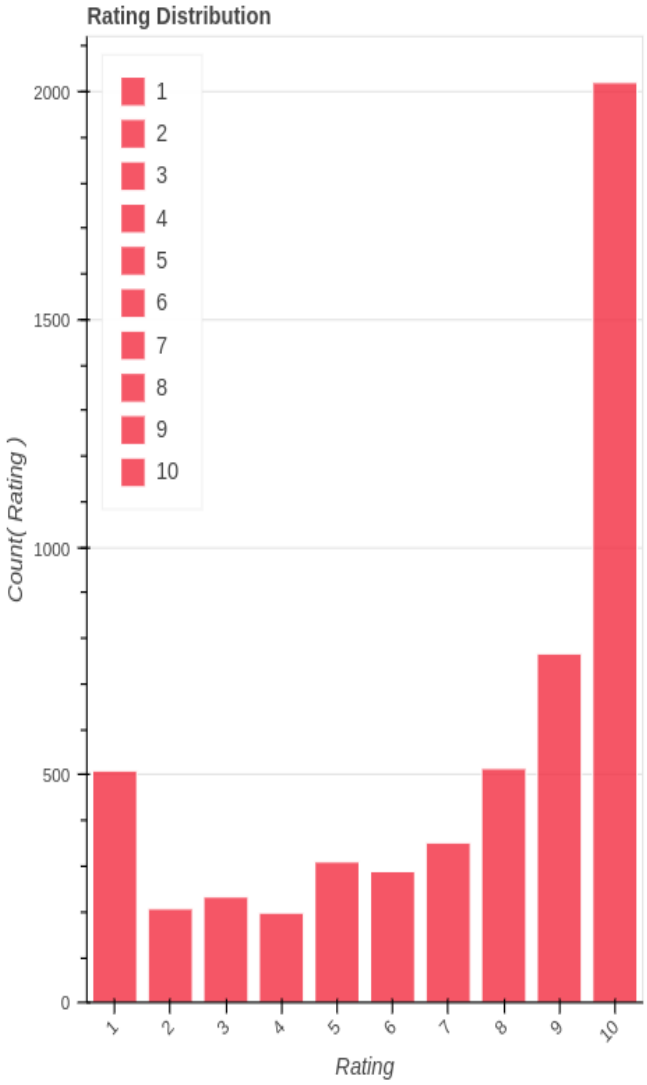

Illustration 3: Rating Distribution

As we will see in future sections, having a small and unbalanced dataset require a special treatment such as using cross validation and stratification. In this way, the weaknesses of the dataset can be surpassed.

Feature vectors and Dimensionality

In supervised learning, we cannot fit entire sentences into a model directly, we have to transform this set of sentences into **feature vectors**. As it will be explained in the **Data Preprocessing section**, to solve this problem we will used bag of words approaches.

Bag of words is a simple and effective feature extracting technique in text processing for transforming texts into **feature vectors** [8]. We can map each text with its equivalent vector of words, which represents the words contained in the text. All the unique words found in the whole set of texts make up the **vocabulary** for all the texts. The size of this vector is defined by the size of the vocabulary, and it is the same for all texts. So, the bigger the set of sentences, the bigger is the underlying vocabulary.

From the images below, it is possible to see how the vocabulary (feature vector dimension) expands by the number of texts being analyzed in the dataset:

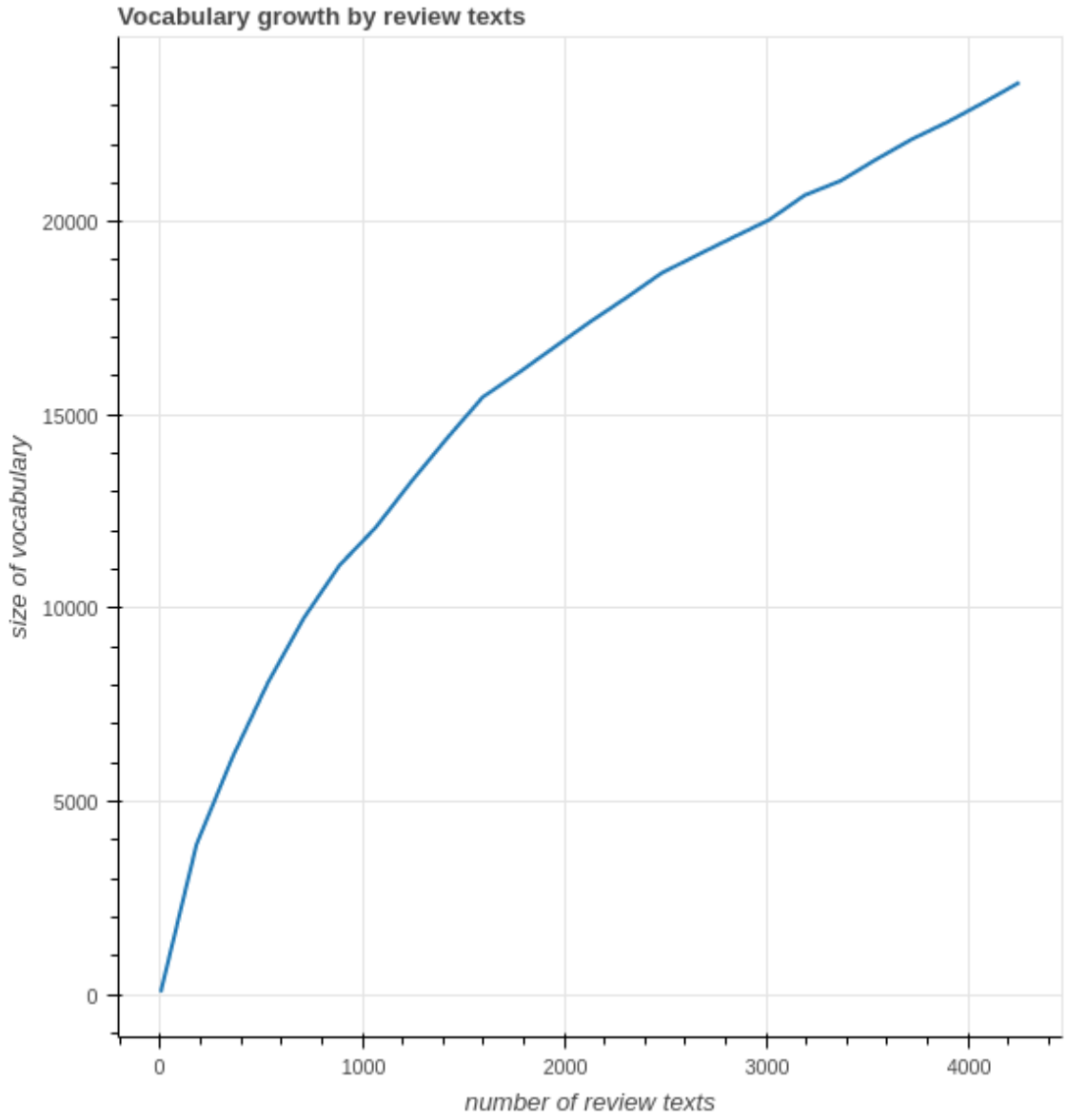

Illustration 4: Vocabulary growth by review texts using CountVectorizer with stop words removed

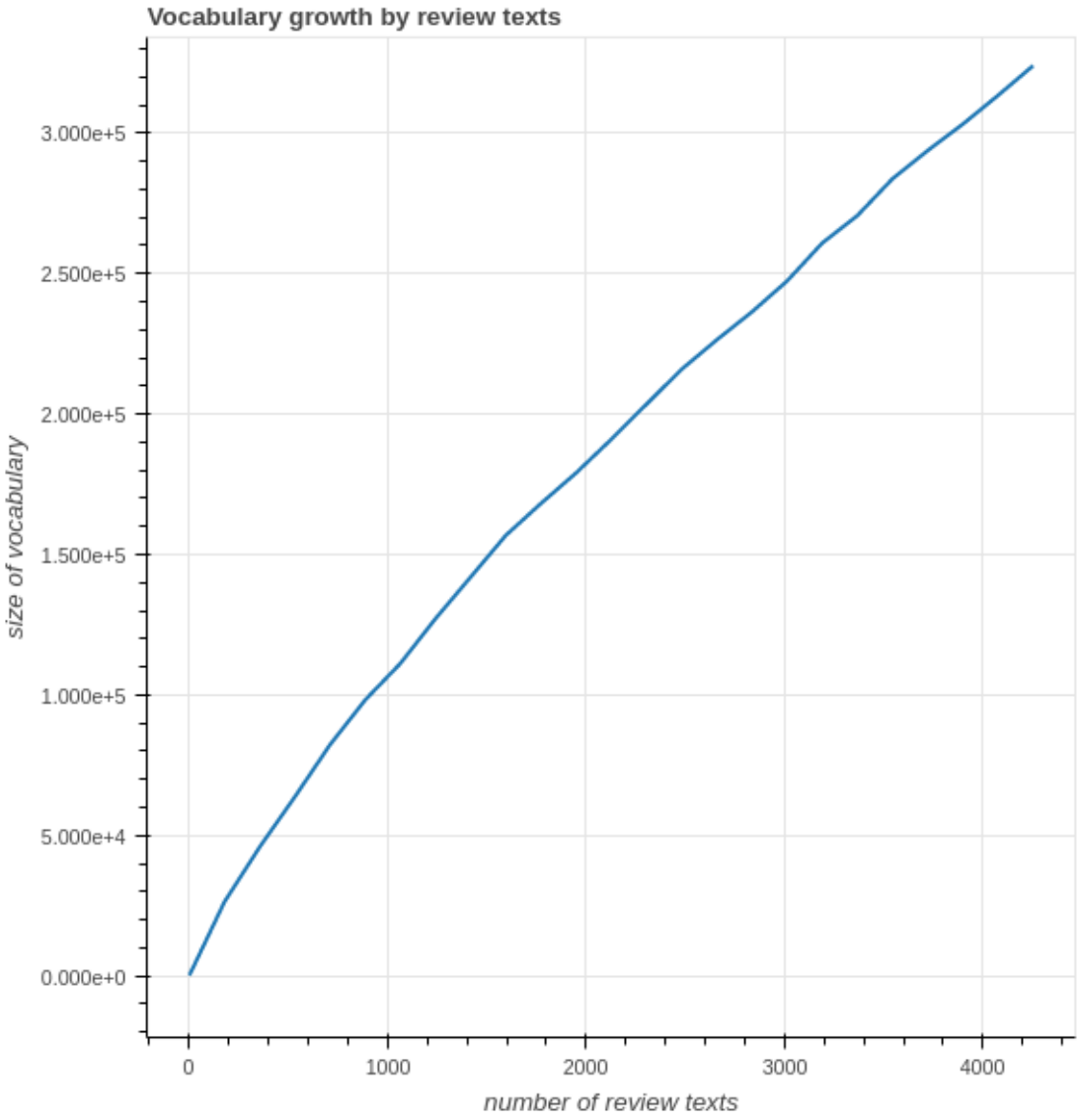

Illustration 5: Vocabulary growth by review texts using CountVectorizer with ngram_range set to (1, 2)

The details of bag of words along with its customization will be explained in the next section. However, what is worth considering here is that this technique has a major drawback, it is memory-intensive. As the dimension of the vocabulary increase, fitting feature vectors into memory becomes harder. Even for small datasets, much memory is demanded.

A solution for this memory problem is to use hashing techniques to limit the size of the feature vectors. So, we decrease the memory cost at the expense of decreasing overall accuracy, a common tradeoff that machine learning engineers face.

Algorithms and Techniques

A number of classifiers can be used in text processing, and the ones that work best will depend not only on the characteristics of the model, but also on the data that is used. And as we have seen it in the **Data Exploration section**, the data collected is **small** and **unbalanced**.

The classifiers chosen to solve the problem under discussion can be divided in the following groups: Linear Models, Support Vector Machines, Naive Bayes, and Decision Trees. The specific models used will be described in the **Implementation section**.

Linear Models

Linear Models are used to divide data that is linearly separable, which can be divided into different classes by hyper-planes. These models are faster than other algorithms, and represent the simplest solution for the case. Even being harder to prove that the data is linearly separable, they can be used to assess their performance, where higher scores mean linear separability.

Support Vector Machines

Support Vector Machines (SVM) is a popular technique for text classification problems delivering great results in the field [16]. There is a solid theory concerning margins that backs up SVM, and which gives a reliable guarantee regarding overfitting. As a result, even for small datasets, which is the case, the model works well and is less prone to overfit than other models.

One major drawback in using Support Vector Machines is that it is memory intensive. SVM can be very inefficient for large data sets. Another downside is that there are too many parameters to tune: *kernel*, *gamma*, *C*, etc., being harder to find the optimal combination for them.

SVM is a great match for this problem because it is less prone to overfit and it is not too costly since the data is small. Besides, as we don't know if the problem is linearly separable, using SVM gives us the opportunity to try different **kernel** functions, like *rbf* and *linear*.

Naive Bayes

Naive Bayes is famously used in spam filtering, where the goal of the model is categorize new e-mails as spam or not [17]. Its convenience comes from being easy to use and set up its parameters. Naive Bayes is based on the naive

independence assumption between its features, in other words, every feature is independent from one another.

Naive Bayes can be really fast compared to other supervised learning models, and it doesn't need high amounts of data to work properly. Although, it has its best performance in situations where there is a weak correlation between the underlying features.

Naive Bayes is a high bias model. In other words, if the naive independence assumption doesn't hold, the model can perform poorly. That's the case when the problem has too many features to consider, or when they are strong related to one another.

Naive Bayes usually works well in practice, even with the over-simplified assumptions about its features. So, even if some of the underlying assumptions don't hold in this project (high number of features and dependence between them), it can turn out to work well. For that set of reasons, Naive Bayes is an excellent candidate for the analysis.

Random Forest

Random Forests is an ensemble method based on Decision Trees. It can represent other than linear relationships between features. Basically, it combines the results of several Decision Tree estimators independently to generate a better combined estimator.

One of the major weaknesses of Random Forest is that it is easy to overfit the data. This usually happens when we don't have enough data to extract the underlying relationships between features, or when the generated trees grow too much (too many branches).

Benchmark

As a benchmark for this project, we will use the performance of Random Forests when comparing with the other models.

For being a high variance model, thus flexible in a number of situations, Random Forest will be used as a benchmark for this project. Besides, as we have no clue if the problem is linearly separable, the model will give us a lot of flexibility.

III. Methodology

Data Preprocessing

Text Feature Extraction

As initially explained in the **Data Exploration section**, it is necessary to transform raw texts from reviews into features that can be fit directly into the algorithms. This process is known as feature extracting.

Defining the vocabulary that will be used from now on, the whole set of texts is known as the **corpus**, and texts can also be referred as **documents**. Bag of words was the chosen feature extraction method to transform each document in the corpus into a feature vector.

Bag of words can be represented by three major steps. For the simplest representation, these steps are [9]:

1. **Parse** documents into tokens by using white-spaces and punctuation as token separators, and associate an integer index with each **unique** token in the corpus. The set of tokens found in the texts makes up the **dictionary** of the corpus.

2. There is a feature vector associated with each document. In order to build this vector, for each document, **count** how many times each word of the vocabulary occurs in it and **assign** the value to the word's index.

3. **Normalize** and **weigh** the assigned value depending on its distribution among the documents.

In most of situations, tokens are represented by single words. However, tokens can also be an **ngram**, which is composed by two or more words in sequence. It is worth saying as well that, because the size of the vocabulary is way bigger than the words in a document, the feature vectors generated are usually **sparse**.

For the implementation of a simple bag of words, we will use the class ***CountVectorizer*** from *Scikit Learn* [11]. For this simple representation, the normalization step is skipped.

Stop Words

When parsing texts, there are a lot of words that don't add any meaning to final analysis. These can be articles, prepositions, pronouns, or even overused words in the set of review texts. They are known as stop words, and can be left out from the analysis [10].

Using CountVectorizer, we can suppress stop words from the final vocabulary., either by providing a list of stop words for the *stop_words* parameter or assign it to *english,* which will remove stop words of the English language.

Removing stop words is helpful to make the vocabulary smaller and more meaningful than if it was created without their removal. The process can be seen as a way of decreasing noise in the data, leaving only the most important words for analysis.

Ngrams

In some cases, words have different meanings when they are next to other words. Single words in a bag of words cannot represent this situation. So, one way to solve this problem is to use, instead of just one word (unigram) to represent a token, two (bigrams) or more words (**ngrams**).

For example, consider the sentence “movie not good” [12]. Seeing the word “good” in a simple bag of words approach would likely assign a positive sentiment to the sentence. However, it is clearly wrong. If we were to use **bigrams** instead, “not good” would be mapped into the vocabulary, and would be assigned as negative.

However, this solution requires more memory when using both unigrams and bigrams. As bigrams are more difficult to be repeated in the documents, the vocabulary of the corpus would tend to grow at a faster rate. In *Scikit Learn*, ngrams can be tuned through the *ngrams* parameter.

TF-IDF

Another technique used to extract more meaningful information of the text reviews is called TF-IDF (term frequency–inverse document frequency). It acts in the third step of the bag of words (normalization).

It weighs each token according to two metrics: the frequency of the word in the document (**term frequency**) and the frequency of the word in the corpus (**document frequency**). TF-IDF value increases the more times a term appears in the document, and decreases the more times a term appears in the corpus.

The important thing to keep in mind is that the TF-IDF tends to assign higher values to important and meaningful tokens in the feature vectors, and lower values to common tokens across the set of documents. *Scikit Learn* offers the option of including TF-IDF in the preprocessing pipeline by using ***TfidfTransformer*** [13] or ***TfidfVectorizer*** [14].

Feature Hashing

As said previously, the number of features generated by bag of words can be huge, specially for large datasets. So, fit all the feature vectors into memory can be memory expensive. A solution for this problem is to reduce the feature vector size to a smaller number of features by using **feature hashing**.

Instead of having a mapping from tokens to feature indices, which is loaded in memory, feature hashing uses a hashing function to this mapping. In this way, to know the position of each token, the token string is passed as a parameter to the hashing function, which then outputs the token's index. The generated index will be mapped in a array with size given as a parameter (*number of features*).

Using the hashing function comes with an associated cost: there is the possibility of collisions (two tokens with the same hash), and there isn't the inverse transformation (hash to token).

In *Scikit Learn*, we can perform this hashing transformation by means of ***HashingVectorizer*** [15]. The *n_feature* parameter represents the number of features (size) of the feature vectors.

Preprocessing Workflow

For the bag of words structure of this project, there are only two decisions users can make to build it. The first is to choose whether to use feature hashing or not, and the second is the usage of TF-IDF for normalization.

Depending on the settings the user want for his analysis, a pipeline can be defined by combining the parameters explained above. The figure below shows how these components can be put together:

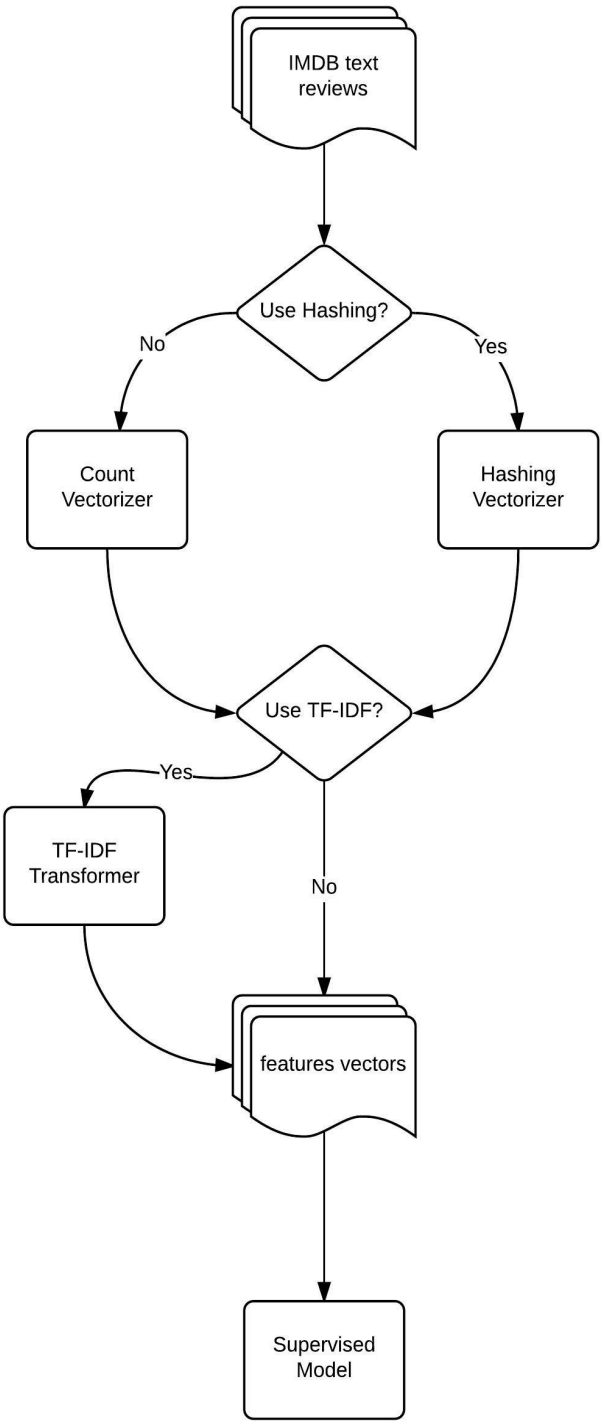

Illustration 6: Preprocessing workflow

Implementation

Linear Models

1. Logistic Regression – model that measures the relationship between the features and the target class by estimating probabilities [18]. It is used the class *LogisticRegression* from *Scikit Learn* [19] with its default parameters set: *penalty='l2' and C=1.0.*

Support Vector Machines

1. Linear Support Vector Classification – variant of the general Support Vector Classifier (SVC) with the *kernel* parameter set to *linear* [20]. Default parameters used: *penalty='l2' and C=1.0.*

2. Support Vector Classification – standard SVM implementation, and it was used the *SVC* classifier in *Scikit Learn* [21] with its default parameters set: *C=1.0, gamma='auto' and kernel='rbf'*

3. Nu-Support Vector Classification – similar to SVC but uses a parameter to control the number of support vectors [22]. Default parameters used: *nu=0.5, gamma='auto' and kernel='rbf'*.

Naive Bayes

1. Multinomial Naive Bayes – implements Naive Bayes according to the multinomial distribution [23]. The class MultinomialNB from *Scikit Learn* [24] is used with its default parameters: *alpha=1.0.*

2. Bernoulli Naive Bayes – implements Naive Bayes according to the multivariate Bernoulli distribution [23]. The class BernoulliNB from Scikit Learn [25] is used with its default parameters: *alpha=1.0.*

Random Forest

1. Random Forest Classifier – set of decision trees classifiers using the class *RandomForestClassifier* from *Scikit Learn* [26] with the default parameters: *n_estimators=10 and min_samples_split=2.*

Refinement

Cross Validation and Stratified K-Fold

A general approach to avoid *overfitting* is to divide the dataset in two parts for different purposes: one for training the model, and other for testing. The first part is known as the **train set** and the second as **test set**. In this way, the model is trained only by the train set, and tested by its test set. Therefore, we can guarantee that the trained model can generalize to new unseen data, and the test score is reliable. In *Scikit Learn*, the class *train_test_split* [28] does the task described above.

As the dataset is small, dividing the data in train and test sets makes these datasets even smaller. We can solve the problem of training properly small datasets by using **cross validation**. The basic approach called **K-Fold** consists of splitting the train set in *k* folds, and for each fold uses *k – 1* to train the model and the selected one for validation.

However, once the dataset is unbalanced, it's important to consider its skewness when splitting in *k* folds. We can do it in a way that the proportion of the target class in each *k* folds be the same,this is known as **Stratified K-Fold**.

When splitting the data into train/test set, we can make a stratified division by using the *stratify* parameter from *train_test_split.* In the cross validation, we can use the class *StratifiedKFold* from *Scikit Learn* [28] to achieve this goal.

Grid Search Cross Validation

Each algorithm has a set of parameters that can be adjusted during cross validation. Depending on the value each parameter assumes, it can improve or worsen the general model performance. From the list of models, here are some parameters that be tuned:

- **LogisticRegression**: *C=np.logspace(-1, 3, 5)*, which **np** stands for the numpy library and ***logspace****(-1, 3, 5)* represents the list [*1/10, 1, 10, 100, 1000*].
- **LinearSVC**: *C=np.logspace(-1, 3, 5).*
- **SVC**: *C=np.logspace(-1, 3, 5), gamma=np.logspace(-4, 0, 5).*
- **NuSVC**: *nu=np.linspace(0.1, 1, 10), gamma=np.logspace(-4, 0, 5).*

- **MultinomialNB**: *alpha=np.linspace(0.1, 1, 10).*
- **BernoulliNB**: *alpha=np.linspace(0.1, 1, 10).*
- **RandomForestClassifier**: *n_estimators=(1, 10, 100), min_samples_split=(2, 5, 10).*

In order to search for the optimal combination of parameters in a given algorithm, we can use the class **GridSearchCV** [29]. It receives as parameters the estimator and the values each parameter can be assigned to, then it performs an extensive combination between these parameter values.

Randomized Search Cross Validation

If there is a large number of combinations, trying out all the combinations can be time-consuming. In order to solve this problem, we can use the class **RandomizedSearchCV** [30] instead. It implements a random search over parameters, and tries out a sample of the combinations. The number of iterations can be assigned by the *n_iter* parameter.

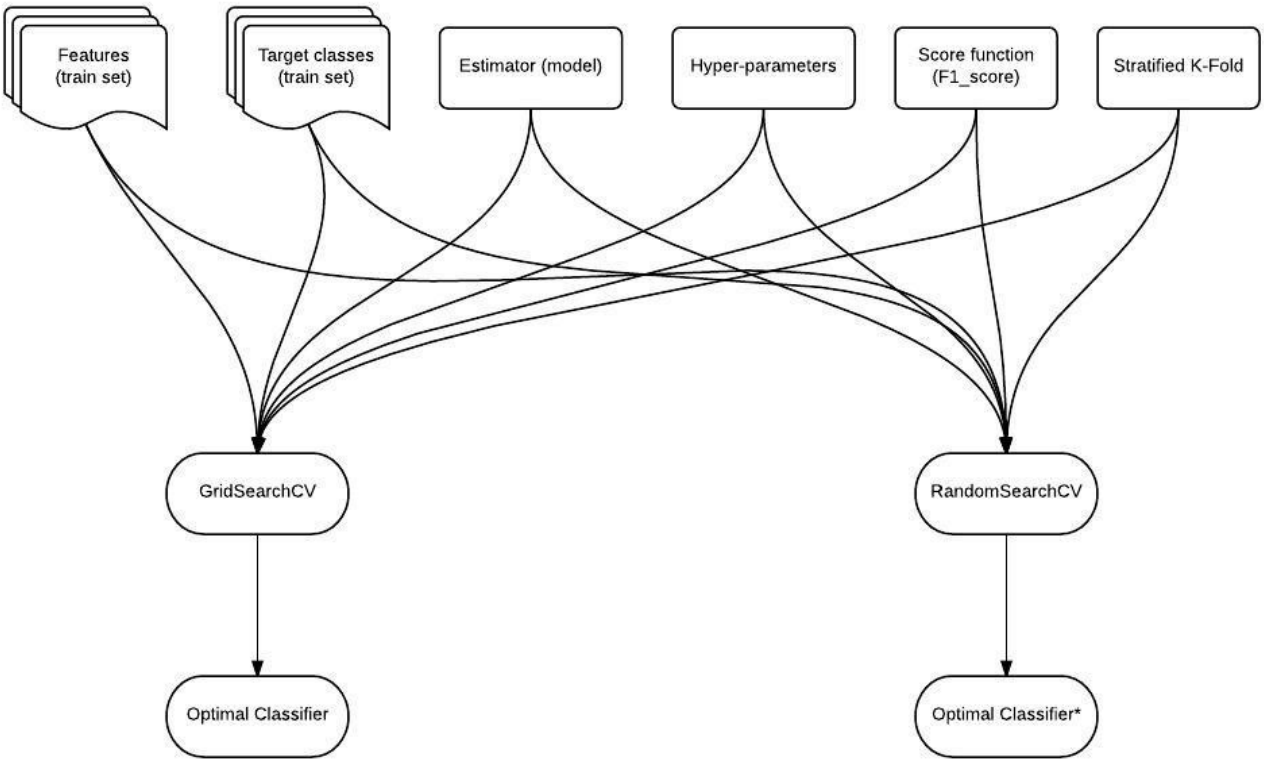

Illustration 7: Refinement workflow

The whole process of refinement can be summarized by the above workflow.

IV. Results

Model Evaluation and Validation

Initially, there are four different preprocessing pipelines, which are the following:

1. Preprocessing Pipeline I = [CountVectorizer]
2. Preprocessing Pipeline II = [CountVectorizer, TfidfTransformer]
3. Preprocessing Pipeline III = [HashingVectorizer]
4. Preprocessing Pipeline IV = [HashingVectorizer, TfidfTransformer]

Running the models in these pipelines, we have four different settings. Pipelines 1 and 2 define a simple bag of words and another using TF-IDF. Pipelines 3 and 4 represent a bag of words using feature hashing and another using feature hashing with TF-IDF. The latter group represents an environment where the memory resources are limited, while the former does the opposite. The best algorithm should output good scores compared to its competitors independently of the environment.

Simple bag of words

The simplest bag of words representation is made up of just the **CountVectorizer**. However, some parameters can still be tuned in this approach, mainly by removing **stop words** and using **ngrams**.

We will use two values for the *stop_words* parameter, either *None* (not removing stop words) or *english* (remove english stop words from text). For ngrams, we will limit the analysis for unigrams (*ngram_range* parameter equals to *(1, 1)*), and unigrams + bigrams (*ngram_range = (1, 2)*).

Combining the *stop_word* values with the *ngram_range* values will create four different settings for this pipeline, as it is shown in the table below:

Model	*stop_words = None, ngram_range = (1, 1)*	stop_words = *None*, ngram_range *= (1, 2)*	stop_words = '*english*', ngram_range *= (1, 1)*	stop_words = '*english*', ngram_range *= (1, 2)*
Logistic Regression	0.945	0.947	0.948	0.945
LinearSVC	0.945	0.947	0.945	0.946
SVC	0.939	0.946	0.944	0.944
NuSVC	0.938	0.946	0.944	0.944
MultinomialNB	0.945	0.938	0.939	0.931
BernoulliNB	0.945	0.930	0.945	0.915
RandomForest Classifier	0.910	0.904	0.909	0.907

Table 1: Pipeline = [CountVectorizer]

Analizing the above results, it is possible to see that the performance among the models and settings are very similar, except for the Random Forest which is below average (benchmark).

Simple bag of words with TF-IDF

Performing the same analysis of parameters as above:

Model	*stop_words = None, ngram_range = (1, 1)*	stop_words = *None*, ngram_range *= (1, 2)*	stop_words = '*english*', ngram_range *= (1, 1)*	stop_words = '*english*', ngram_range *= (1, 2)*
Logistic Regression	0.952	0.950	0.949	0.943
LinearSVC	0.954	0.953	0.949	0.948
SVC	0.949	0.953	0.950	0.948
NuSVC	0.951	0.957	0.950	0.946
MultinomialNB	0.913	0.884	0.922	0.884
BernoulliNB	0.945	0.930	0.945	0.915
RandomForest Classifier	0.901	0.899	0.915	0.912

Table 2: Pipeline = [CountVectorizer, TfidfTransformer]

As we can see above, the best performance comes from Linear Models and Support Vector Machines. Bernoulli Naive Bayes has a decent performance as well. Multinomial Naive Bayes does not perform well because the feature vectors are float numbers [24].

Bag of words with feature hashing

Using the HashingVectorizer in the pipeline, and limiting the dimension of the feature vector by *5000*, we have the following results:

Model	*stop_words = None, ngram_range = (1, 1)*	stop_words = *None*, ngram_range *= (1, 2)*	stop_words = '*english*', ngram_range *= (1, 1)*	stop_words = '*english*', ngram_range *= (1, 2)*
Logistic Regression	0.944	0.936	0.941	0.935
LinearSVC	0.942	0.939	0.940	0.935
SVC	0.948	0.939	0.938	0.932
NuSVC	0.948	0.939	0.941	0.932
MultinomialNB				
BernoulliNB	0.919	0.923	0.919	0.897
RandomForest Classifier	0.894	0.890	0.917	0.898

Table 3: Pipeline = [HashingVectorizer]

Again, the best performance comes from Linear Models and Support Vector Machines, and for the same reason above, Multinomial Naive Bayes does not perform well. Using *ngram_range = (1, 2)* leads to slightly worse results than *ngram_range = (1, 1)* because when we parse the text into tokens, they will be mapped to a feature vector of size 5000. So, the more tokens are mapped to the feature vector, which is the case for *(1, 2),* the more colisions will happen (worse performance).

Bag of words with feature hashing and TF-IDF

Model	*stop_words = None, ngram_range = (1, 1)*	stop_words = *None*, ngram_range *= (1, 2)*	stop_words = '*english*', ngram_range *= (1, 1)*	stop_words = '*english*', ngram_range *= (1, 2)*
Logistic Regression	0.952	0.937	0.940	0.934
LinearSVC	0.950	0.938	0.938	0.931
SVC	0.947	0.935	0.938	0.925
NuSVC	0.948	0.933	0.938	0.929
MultinomialNB				
BernoulliNB	0.919	0.923	0.919	0.897
RandomForest Classifier	0.898	0.884	0.918	0.899

Table 4: Pipeline = [HashingVectorizer, TfidfTransformer]

Again, Linear Models and Support Vector Machines stand out among the other models using this pipeline.

It is worth noting that using or not feature hashing does not make almost any difference in performance. Since feature hashing leads to less memory allocation, we will define as the default preprocessing pipeline the usage of feature hashing and TF-IDF (**Pipeline IV**).

Removing stop words and using unigrams will lead to a smaller vocabulary than the other settings (see Illustration 4 in Data Exploration section). Since the values for this settings don't influence much the performance of the models, ***stop_words*** **= '*english*'** and ***ngram_range = (1, 1)*** will be set as the default parameters for feature extraction.

Finally, as we can see from the tables above, Linear Models and Support Vector Machines have similar results. Because Support Vector Machines is compute-intensive even for small datasets [21], **Logistic Regression** was chosen as a preferred method for the analysis. Linear Models are much simpler to implement and faster to run. In addition to that, due to the Occam's Razor principle [31], the simplest model should be picked among some with similar performance.

Justification

Comparing the results for Logistic Regression to the benchmark (**Random Forests**), there is quantitative improvement in the F1 score. The difference in performance ranges from 2% to almost 6% depending on the preprocessing pipeline used. It is worth considering that both models are using GridSeachCV to search for their best hyper-parameters. Achieving an F1 score of approximately 0.95 in most of the cases shows that the final algorithm is a very good fit for tthe problem under discussion.

V. Conclusion

Free-Form Visualization

Sentiment analysis can be applied in a number of fields, it is specially important for tracking metrics online. It can be used in social medias to track company's popularity among users, helping the system spot some potential problems with products and services. For example, companies can use sentiment analysis to know their reputation among users, and finding ways to improve it. So, they can act more proactively since they can find problems faster.

Applying sentiment analysis to the financial world to forecast market movements can help investors to increase their profits. A lot of resources can be tracked: blogs, news, social medias, and so on. Due to noise in the data, the algorithm can misinterpret the information. So, in most of these situations, the predictive analysis acts as a filter to investors as a way to show them possible market variations.

There are a number of examples showing the relationship between stock prices and sentiment analysis in this link reference [32]. It is shown below a graphic comparing Microsoft stock price and a sentiment index called Sentdex:

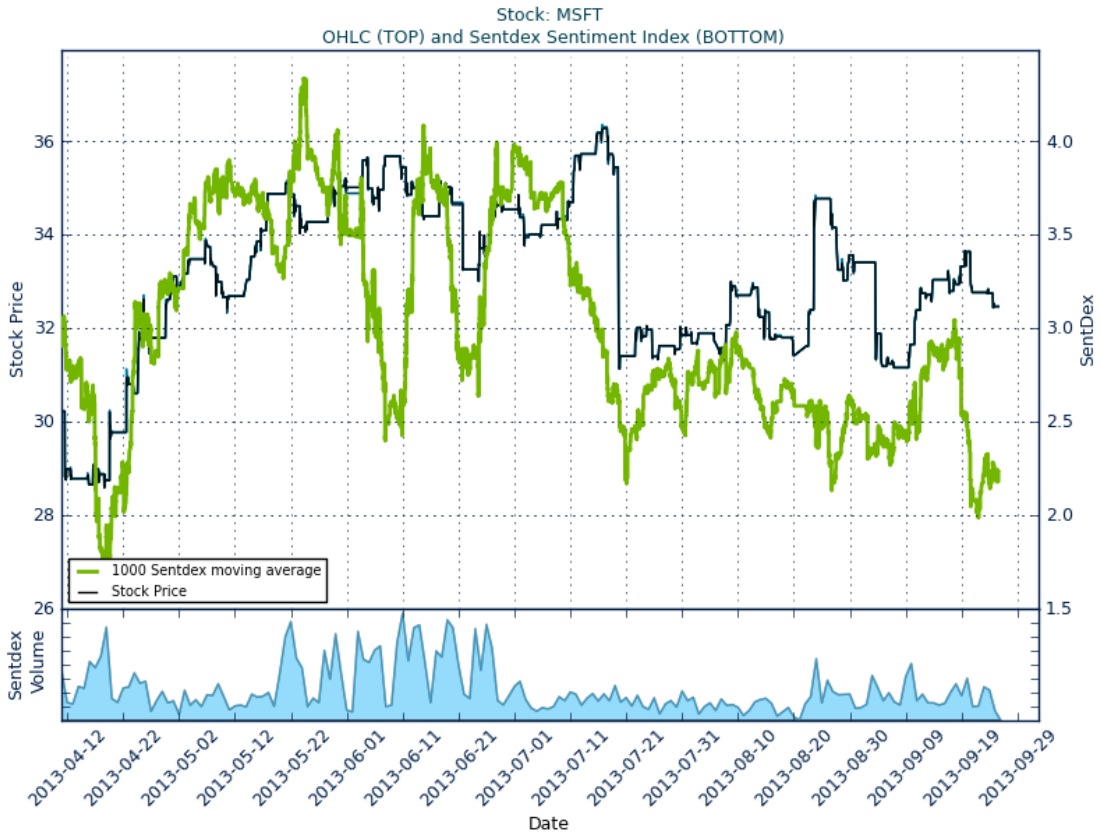

Illustration 8: Sentimet Analysis vs Microsoft Stock Prices

Taking advantage of this sentiment analysis, movies and series makers can tracked the popularity of their creations. Being able of processing accurately online data from their users will put them in a better position than their competitors. Acting in a proactive way when facing bad reviews from series, TV shows, movies, etc. will give the industry an instant feedback about their work, and the opportunity of changing the course of their product if so. The opportunities are endless.

Reflection

Seeing the project from a broader perspective, we can divide it into important parts: getting, interpreting, and manipulating the data, and extensively testing the models.

Selecting the data to be used for text processing could be as broad as possible. Instead of using reviews for crime series in the analysis, other categories could be used for the analysis as well. Moreover, two important ranges of rating stars were defined to represent negative and positive sentiments in reviews, [1, 3] and [8, 10], respectively. If we were to choose broader ranges, we would likely achieve lower performance since it would be harder to distinguish negative from positive reviews.

Another part of the project was to analyze the characteristics of the data in order to adapt and adjust it. The downloaded data is very small and unbalanced, requiring the usage of cross validation and Stratified K-Fold.

Then, it was provided a general approach to transform texts into feature vectors for supervised learning models, the bag of words. When training and testing the models with the data, a number of algorithms could be used for the project. Finally, a search for the best combination of parameters in each algorithm was taken place to find the best performance.

Improvement

For feature extraction, it was used simple implementations of bag of words. However, there are other techniques, which have better performance depending on the text structure. Using Neural Networks, it is possible to extract feature vectors from texts [33]. In that way, the part of feature extraction can still be further developed in this project.

Moreover, the simple fact of using feature vectors, the words are considered in isolation. We do not consider the order in which they appear in the text review.

There are situations where single words express one sort of sentiment, but the whole sentence expresses the opposite.

Because the order is a valuable source of information for sentiment analysis as well, it is important to consider other approaches than feature vectors. For this kind of analysis, it is necessary to go beyond and use Deep Learning to help build the representation of whole sentences [34].

Nonetheless, for a preliminary analysis using text processing in the context of review websites, such as IMDB, the study that was developed shows practical and useful solutions for the problem.

Reference

[1] https://www.lexalytics.com/technology/sentiment

[2] http://www.scholarpedia.org/article/Text_categorization

[3] http://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html

[4] http://www.imdb.com/

[5] http://docs.python-requests.org/en/master/

[6] https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[7] http://www.imdb.com/search/title?genres=crime&title_type=tv_series,mini_series

[8] http://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction

[9] http://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction

[10] http://xpo6.com/list-of-english-stop-words/

[11] http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

[12] http://fastml.com/classifying-text-with-bag-of-words-a-tutorial/

[13] http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html#sklearn.feature_extraction.text.TfidfTransformer

[14] http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html#sklearn.feature_extraction.text.TfidfVectorizer

[15] http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html

[16] http://nlp.stanford.edu/IR-book/html/htmledition/support-vector-machines-and-machine-learning-on-documents-1.html

[17] http://scikit-learn.org/stable/modules/naive_bayes.html

[18] https://en.wikipedia.org/wiki/Logistic_regression

[19] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[20] http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

[21] http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[22] http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVC.html

[23] http://scikit-learn.org/stable/modules/naive_bayes.html#multinomial-naive-bayes

[24] http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html

[25] http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html

[26] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[27] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

[28] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

[29] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[30] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

[31] https://en.wikipedia.org/wiki/Occam's_razor

[32] http://sentdex.com/how-accurate-is-sentiment-analysis-for-stocks/

[33] https://code.google.com/archive/p/word2vec/

[34] http://nlp.stanford.edu/sentiment/