A Report on

Opinion Spam Detection using Language Models and Naïve Bayes

Group Name - “Amateurs”

Group Members - Rajdeep Ghosh – RG677, Sushmitha Vemula – SV482

**Abstract:**

Advances in the fields of natural language processing have been assisting us in detecting any kind of fake emails and reviews, sentiment analysis and many other such applications. With such advancements in mind, we started off working on the simple opinion spam detection problem. There are several approaches in solving this problem like using the language models (Unigram, Bi-gram and n-gram models) or supervised machine learning algorithms. This project aims to compare the performance of the language models like Unigram and Bigram alongside implementing several Unknown word handling and smoothing techniques. In addition to the language models, we implement the Naïve Bayes (supervised ML algorithm), experimenting with several types of Naïve Bayes (Bernoulli and Multinomial NB) and linguistic features. We find the Bernoulli perform well with the small vocabulary, with slight (3-4%) difference in the accuracy levels of Bernoulli and Multinomial NB.

**Introduction:**

In a world where we talk and ask things to internet than any other human around, problems like spam-detection in any business area will be immensely impactful. When we try finding a good restaurant/ hotel around us, we tend to go read the reviews online. In such cases, fake reviews either positive or negative ones will impact the choice we make. From the business perspective there will be huge losses because of these fake reviews.

In order to mitigate these problems, we have worked on a few modelling techniques like Language models (unigram and bigram) and a supervised machine learning algorithm like Naïve Bayes classifier for assisting the businesses to classify the true and fake reviews and in turn act on them. In the Naïve Bayes technique, there are various categories which will help in the effective classification of the text – Like Multinomial Naïve Bayes and Bernoulli Naïve Bayes. In this project, we have implemented both the techniques with different levels of smoothing and found that the Bernoulli NB at Alpha=1 is the most effective one for this data set.

The data set provided for this classification problem is the set of observations which have the reviews of 20 different hotels in Chicago. The data collection method is mentioned in detail in this [paper](https://www.aclweb.org/anthology/P11-1032).

**About the Data:**

The data set is the list of truthful and deceptive hotel reviews of 20 different Chicago Hotels. The entire data set is divided into text files one with truthful reviews and other with deceptive. The text files are organized in such a way that the reviews are in the form a string and each review is separated from the other using a new line character. We are given the training data set with 512 different reviews for each these deceptive and truthful documents, around 125 reviews in validation set and 320 reviews in the test set.

The entire data set as required for the implementation of all the models is divided into Training, validation and test data sets. The training set is used for training of the model. The validation set is also in the same format with the truthful and deceptive reviews provided separately. This can be used as the labels of the reviews and can be used for the validation of our model built. The test data is a single file for which we find the label using the classifier.

In the data set provided, the tokenization is already done for all the files. Though the tokenization is done already, we have seen a lot of anomalies in the data set. If we take the simple example provided above, the truthful review, we can see that there are spaces between “was” and “n’t” when it has to be considered as a single word all together. In addition to this, in the same sentence we see a price tag where the symbol $ and the number is again separated by space. In such cases, the model treats them as two different tokens which might not be quiet right as sometimes numbers combined with dollars can be a good indicator of the fake and truthful reviews.

Also, there are many other anomalies that we have encountered like extra space added where it is not required. Like ‘s, ‘ll, ‘d, n’t, ‘m, etc. In such cases where the apostrophe and the letter has to be combined with the word is separated and treated as a different token in the model.

Finally, there are also instances, where the same word for example received is written in different cases at different positions of the file leading to treating these as different tokens which is not quiet right in many cases.

**Example**: received, Received, RECEIVED

Although there are instances where some punctuations are treated as separate token, it can still be useful as we have seen that one of the training data sets had “!” as the token with highest frequency but that’s not the case with the deceptive file. So, we have considered this as a indicator token to differentiate both.

However, Python provided a lot of text cleaning and **NLTK** functions and packages for cleaning of the data for effective tokenization. Examples are **NLTK stop\_words, textcleaner package, lemmatization, stemming functions.**

In this project, while working on these data sets, though it is said that the tokenization step is completed before hand, we have explored various data pre-processing methods for cleaning the data before we input it to the model. We have tried removing the stop words, converting the entire file to lower case and making the tokens based on that, removing the numbers and only considering punctuations and letters, removing the spaces for the above -mentioned examples and also performed stemming using Porter Stemmer. But we later found out that basic pre-processing which included converting the characters in the reviews to lowercase yielded best results.

**Implementation:**

**Data-Preprocessing and Packages**

We used a function **lower()** for both Language models and Naïve Bayes to covert to lowercase. Also, we defined a function individual words which would take the file as input argument, and splits it into tokens, and then stores them review by review in a list. In this manner, we were able to prepare both the truthful corpus and the deceptive corpuses from the training data.

1. **Language Models – UniGram and BiGram Implementation**

We have experimented with the simplest type of model which are N-gram models.

**Packages Used for Implementation in Python:**

Numpy – Used for calculating the log probabilities and the perplexity calculation

Pandas – For exporting the final results data into a CSV file

NLTK - For cleaning or pre-processing of the data (stop words, Lemmatization and Stemming)

**Data Structures Used:**

**Dictionaries** - as they were easy to access and stored probabilities(values) according to the n-grams(keys).

**Lists** - to store tokens.

**Unsmoothed n-grams:**

Starting off with the simplest type of n-gram, which is the Unigram. To get the unigrams, we built two functions, Unigram and UnigramCount. The Unigram function would take the aforementioned lists of reviews as input, create a **dictionary** to store the unigrams as keys, and their counts in the file as values of the dictionary. If the unigram is not present, we assign the initialize the count of it as 0 and then increment it by one whenever we would come across the word again while parsing through the file. We then return the complete dictionary containing the unigrams with their counts. UnigramCount counts the number of each token and returns the total number of Unigrams.

Next, we have the 2-gram model or the Bigram Model. The implementation was a bit similar to Unigrams, but the catch is that now we have to compute pairs of words and their counts. For which, we created two function, Bigram and BigramCount. In a similar manner, Bigram function would take a list of review(token-wise) as input, create a new **dictionary**, and store the pairs of words and the next words, until the end of the review has been reached. This is done for all reviews using a loop. Again, it is checked if the Bigram is present in the dictionary or not. If no, then it has an initialized count of 0 in the dictionary with an increment of 1 whenever we come across it later through the parsing stages of the file. BigramCount returns the sum of all bigram counts in the file. Unigrams and Bigrams were computed for both the deceptive and truthful training sets.

**Unsmoothed n-grams Probabilities:**

Now that we have the Unigrams and Bigrams, we’ve defined two functions UnigramProbability and BigramProbability for probability calculation.

We calculate the Unigram probabilities by count(unigram)/TotalCount(N) in the data. We then store them in a dictionary where the keys are the unigrams and the values are the probabilities.

We calculate the Bigram probabilities by dividing the count(Bigram)/count(prefix word in Bigram). We then store them in a dictionary where the keys are the bigrams and the values are the probabilities.

Please note that we used log probabilities here, as probabilities range between 0 and 1, and multiplication of probabilities repeatedly would result in a very small number. Hence, log probabilities take care of it for us. The probabilities are computed for both truthful and deceptive sets.

**Smoothing and Unknown Words Handling**

The smoothing technique that we’ve implemented is called Laplace Smoothing (Add-k smoothing). After experimenting a lot with the value of k giving values such as 0.1, 0.5, 1,0.01 and 0.05, we got the best results when we tried smoothing (k=0.1). We performed the smoothing on unigrams using a function defined by us, AdditiveSmoothing, where the arguments were the value of K, count of unigrams in the data(N), distinct vocabulary words(V), the dictionary of unigrams we created above and the corpus list. To perform smoothing, we added k to the numerator and kV to the denominator. If the unigram is not present in the corpus, we simply just put k in the numerator and N+V in the denominator. We then stored these smoothed unigram probabilities in new **dictionaries** which will be used in the perplexity calculation.

Now let us get into how we handled unknown words. What we did was replace the words in the training data with ‘<UNK>’ based on frequency f. We basically replaced those words with ‘<UNK’> who had frequencies below f. In this manner, we were able to train the model for unknown words ‘<UNK>’ just like any other regular word. So, any word which is in the validation or test set which is missing from the training corpuses, is converted to ‘<UNK>’, its count is added to ‘<UNK>’ in the training corpus. Implementing this helps us compute probabilities for unknown words as well, just like any other word in the data.

**Model Evaluation-Perplexity:**

Now that we have our probabilities ready, we need to assess how well our model works on unseen data. We’ve taken unigrams as the language model for calculating the perplexity. We’ve defined a function Perplexity (Review List, SmoothedDictionaries, n-gramProbabilities). Review List is the list of token we created in the beginning, and SmoothedDictionaries are the updated corpuses after smoothing. We then return the exponential computation of -1/N(number of tokens in each review) multiplied by the negative log of all the unigram probabilities of unigrams present in the review, for each review. n-gram Probabilities is another function which computes the addition of all log probabilities which would be equivalent to the multiplication of all the probabilities as log(A\*B) = log A + log B.

We calculate the perplexity for each validation set, truthful and deceptive. For this, we had to prepare two different training corpuses (truthful and deceptive) with smoothing for each validation set, so that we can compare the perplexities per each validation set review and return the class which had the lower perplexity. We then measured the accuracy of the model on both the validation sets, and got 94.5% on the truthful set and 93% for the deceptive. We then proceeded to do this on the test set as well and then converted the results to a CSV, submitting it on the Kaggle Competition Page.

1. **Naïve Bayes Classifier:**

For implementing the Naïve Bayes as well, we have considered both the data existing tokens and also converting the file to lower case. After the conversion, we have combined the truthful and deceptive text file reviews into one huge **data frame** and have assigned a label to each of them as ‘1’ for deceptive and ‘0’ for truthful. The columns of this data frame acted as the input for the CountVectorizer in python (This function takes the raw reviews data as input and converts it into a form understood by the ML algorithm).

**Multinomial and Bernoulli Naïve Bayes:**

In this, we have implemented both Multinomial and Bernoulli Naïve Bayes method and have compared the results at different levels of smoothing values (K=1,0.1). The models also differ in how non-occurring terms are used in classification. They do not affect the classification decision in the multinomial model; but in the Bernoulli model the probability of nonoccurrence is factored in when computing P(C|D). Generally, Multinomial Naive Bayes is in a sense more complex model (we can reduce it to Bernoulli). Because of that, Bernoulli model can be trained using less data and be less prone to overfitting. Sometimes, using the concept of Bias-Variance Tradeoff, simple models are more effective than complex ones.

In simple words, Bernoulli calculates the fraction of documents of class t that contains word t. Also, it used binary occurrences ignoring the number of occurrences. Finally, it is good for small amount of data not for large data sets.

The Time complexity is the same for both Multinomial and Bernoulli NB. So, keeping all these pointers in mind, we have experimented with both Bernoulli and Multinomial NB.

**Model Evaluation:** To evaluate the Naïve Bayes model effectiveness for both Multinomial and Bernoulli, we have used the validation set and the classification results.

**Linguistic features:** To deal with the linguistic features of the document, we have implemented the tokenization techniques mentioned in the Data Set section. (Example – Removing the stop words, converting all the data into lower case and then finding the tokens, Removing the spaces in certain cases). But the results were most effective only for the processing step where we convert every word to lower case.

**Packages Used for Implementation on Python**: **Sklearn.NaiveBayes** for implementing Naïve Bayes, **NLTK** for tokenization and cleaning, Multinomial and Bernoulli NB from **ScikitLearn**, **CountVectorizer** for feature extractions in a form understood by the machine learning algorithm.

**Results/ Observations:**

Firstly, starting with the results seen for pre-processing steps:

* Taking the data as is without any additional changes to the tokens already created yields 88% accuracy on the Unigram model
* Converting all the data in the text document to lower case and finding the token have improved our accuracy to 92% for the Language Models and 91% for the Naïve Bayes classifier
* Also, the tokenization methods where we remove the extra spaces in a few cases (Examples: ‘ll, ‘s , ‘d, ‘m, $ , etc.) haven’t changed the accuracy of either Language Models or Naïve Bayes in this case

Impact of the Smoothing and Unknown Word Handling techniques (Language Models):

* For finding the best way to do the smoothing, we have experimented with different parameters of K = 0.01,0.05,0.1,0.5,1 in the Laplace additive technique and found that the results are in the range of 75% to 92% with K=1 giving the highest. The accuracies mentioned are for the data set where all the data is converted into lower case as mentioned above
* Handling the unknown words by assigning a probability value of 1/V to the unknown word, has not yielded any better accuracy while we classify

Impact of the Smoothing parameters on the results (Naïve Bayes):

* Implementing Multinomial Naïve Bayes with K=1 and K=0.1, which has yielded the highest accuracy at K=1 with 88% classification accuracy
* Similarly, Bernoulli NB implementation at K=1 and K=0.1 has given us the maximum of 91.4% accuracy at K=1

**Kaggle Results**: Team – “**Amateurs**”

Language Model Kaggle Accuracy:

Screen%20Shot%202019-09-30%20at%2012.46.07%20AM.png

Naïve Bayes Model Kaggle Accuracy:

Screen%20Shot%202019-09-30%20at%2012.44.18%20AM.png

**Error Analysis and Comparison of Language Models and Naïve Bayes:**

* We have seen better results for the language models in our case because, the Naïve Bayes model is generally very effective in comparison to language Models when there is a huge amount of data and ton of features. If it’s a small amount of data, the difference in the results will not be of much difference

**Conclusion:**

* Leveraging various language models, smoothing and unknown word handling techniques and Naïve Bayes classifier, we found that the Laplace additive smoothing (Adding +1 to the frequencies) of the Unigram implementation yielded best results for this data set. Also, the Bernoulli Naïve Bayes again with Laplace +1 smoothing have assisted in classifying the reviews into fake and true ones with an accuracy of 91%. With large data sets, we can experiment more with the Multinomial NB and n-gram models with n>1.

**Future Work:**

* As this mainly covers exploring just one technique in the supervised machine learning setting, we can extend this by applying many different supervised machine learning algorithms like Neural Networks, Support vector machines etc. and finding the results of each. Also, instead pf restricting this to just opinion spam detection, we can do the use these language modelling and supervised learning algorithms for problems like sentiment analysis, sentiment parsing, Parts of speech tagging, paraphrase detection, machine translation and speech recognition.

**References:**

* <https://nlp.stanford.edu/IR-book/html/htmledition/the-bernoulli-model-1.html>
* <http://www.cs.cmu.edu/~knigam/papers/multinomial-aaaiws98.pdf>
* <https://books.google.com/books?hl=en&lr=&id=LNkS_tUPHtUC&oi=fnd&pg=PA208&dq=applications+of+NLP&ots=jMukq0tR3v&sig=JPJoZGh9XHbFE4Dok4mEYIVm2hY#v=onepage&q=applications%20of%20NLP&f=false>

**Contributions of the Team Members/ Workflow:**

* There are 2 members in this team of “Amateurs”
* For the first couple of days, we have spent time on brainstorming the pipeline and the different possible ways to implement the project
* After deciding on flow of the project, both of us have started implementing all the models on Python, since both of us wanted to implement the models and compare the results
* This has helped us in finding a better way of implementation

Once the implementation was done, we started making the report by dividing it amongst us in a such a way that the person whose technique had the higher accuracy will write about that particular one (By this, we could divide the report making into approximately 50% each)

**Feedback on the Project:**

* The project has immensely helped us in understanding the concepts better. This project has taught us the importance of making a pipeline before starting our implementation. And the brainstorming work done on the pipeline building and making the model effective, has assisted in a lot of cross-learning activity.
* The project had a difficulty somewhere between medium to hard.
* We have worked on building the pipeline, implementing the models in python, trying out various smoothing and unknown variable handling techniques and finally making the report has taken at least 4 hours of our time every day since the project is handed
* Also, 5-6 projects for the entire semester with no exams will be an interesting one. As we would gain more knowledge in this way.