Review Classification

Balraj Aujla
Cal Poly, San Luis Obispo
College of Engineering
Senior Project
Spring & Fall 2016
December 16th, 2016
Advisor: Dr. Franz Kurfess

Table of Contents

Abstract

The goal of this project is to find a way to analyze reviews and determine the sentiment of a review. It uses various machine learning techniques in order to achieve its goals such as SVMs and Naive Bayes. Overall the purpose is to learn many different machine learning techniques, determine which ones would be useful for the project, then compare the results. Research is the foremost goal of the project, and it is able to determine the better algorithm for review classification, naive bayes or an SVM. In addition, an SVM which actually gave review's scores rather than just classifying a review as negative or positive was created.

The next step is benchmarking both the SVM and Naive Bayes approach. Both algorithms were run through various tests of different input sizes and different reviews. The algorithms were timed for speed in prediction, classification, and most importantly accuracy. Overall, both approaches are great for different reasons that will be explored throughout the project.

Introduction

Classification is a very big problem within the fields of machine learning and AI today. The ability for a computer to determine what something is, can help drive many fields move forward. It helps computer users be more secure from spam attacks, it can help an AI make decisions, it can drive a car, or even help cure cancer and many other diseases. Classification is typically a supervised machine learning technique, which uses a very large data set to predict where the new sample would fall within the provided data set. In the case of this project the classifiers find out if a review is negative or positive, but if fitted with different data the classifiers could do any other kind of prediction.

The original goal of the project was to distinguish between comments that include hate speech and those that do not; however, after finding that no large data set for these comments existed i decided to focus on reviews instead. If a set of comments was found in the same format as the review data set, all classifiers implemented would be able to classify this set and predict hate speech, or any other attribute. Images, audio, or anything other than text can also use the same classification techniques discussed further in the report; however they would have to be implemented slightly differently as words, pixels in an image, or sound bytes have different features.

The following report will discuss the implementations of an SVM and Naive Bayes classifier. These two classification techniques work very well when it comes to text data and achieve about a 95 percent accuracy rating when given a large enough data set. The mathematics and APIs used to implement these two algorithms will be discussed in detail; then the results and performance is tested and analyzed thoroughly.

Technologies Used

- Python - Used to program the project.
- Scikit & SKLearn - API Used to implement many of the machine learning algorithms, such as the SVM, Vectorizer, and Multinomial Naive Bayes.
- Pickle - Used to save the SVM, and Vectorizer so the program will not have to parse entire data set every time it needs to analyze a review.
- Amazon Product Data - A Dataset of 82 Million Reviews, with features such as actual review, summary of review, review helpfulness, and the overall score of the product being reviewed. These reviews have also been aggressively parsed to remove any duplicate reviews.
- Microsoft Excel - Used to create graphs made by benchmarking tests.

Research

The first phase of research was learning about the basic ideas of Natural Language Processing. The first things i learned about was word tokenizing, and a fantastic classification algorithm, **Naive bayes**.

In order for Naive Bayes to work it requires a bag of words representation of the text data. What this means is that all order of the words is lost, and then what remains is a list of the words with a count for each word. For example:

"This product was extremely good, and fun to play! The gameplay was fantastic and extremely fast paced!"

Extremely - 2

And - 2

Was - 2

Fun - 1

…

This is performed over the entire dataset, and the count for every frequent word is recorded. One problem with this representation currently is there are words that do not really mean much. The word "And" is a prime example as this does nothing to reflect if a review is positive or negative, but will most likely be one of the most common words. There are two ways of dealing with such problems, the first being a list of stopwords. The words that are found in both the dataset and stopword list should simply not be added to the bag of words. The only problem with this approach is that it can slow down the program when it needs to check the list to see if it is a stopword for each word it processes. The second approach is much faster, but may result in some mistakes. Essentially, word such as “and”, “the”. “But”, “Or”, etc. are words that appear in most sentences. It is possible to simply set a benchmark to simply remove words that are too common from the bag of words. Of course this can cause problems as the program may inadvertently remove very important and frequent words, or not actually fully remove stopwords.

After the bag of words has been created the rest is left to probability in the naive bayes classifier. Essentially each review will be treated as a document d, and each document will belong to a class that is either positive or negative. The goal of the Naive Bayes Classifier is to a document's class.

Input:

- Document d
- A fixed set of Classes C = {c1, c2, …, c2} //In this example is c1 = positive, c2 = negative.
- A training set (The Amazon Product Review Set) of documents and the class the belong to (d1, c1), … (dn, cm)

Output

- A trained classifier.

The Classifier is built on the following equation, known as Bayes Rule.

$$P(c|d) = (P(d|c)P(c)) / (P(d))$$

We are also able to drop the probability of the document considering the document is a constant, and as we go to maximize this result with different classes dividing by P(d) changes nothing. Therefore the final equation is:

$$\text{Argmax } P(d|c)P(c)$$

$$c \in C$$

$$\text{Argmax } \prod P(x|c)P(c)$$

$$c \in C,\ x \in X$$

(X being vector of features (words) of the Document (Review))

What is the P(c)? It is simply how often a certain class occurs. If positive reviews are in .61 of the dataset, then P(positive) = .61.

What is the P(X|c)? It is simply asking what is the probability X is in the class c. This is where the bag of words assumption gains power. Essentially there is a bag of words for each class, and the probability of x in c is computed. The algorithm determines which class the feature belongs in, and uses the dot product of all features to determine what class the document belongs in.

The specific version of naive bayes used very often within text classification is **Multinomial Naive Bayes.** This merely means that when we predict the probability the algorithm will assume that a multinomial distribution exists, which is very reasonable to assume for word counts; this is because multinomial takes the frequency of words more into account. When calculating P(X|c) instead of just looking at how many times documents within class c contain X, the total count of X across all documents in class c is recorded. This is clearly useful as a document containing two instances of the word awful should be seen as different as a document only including the word awful once.

Another approach for review classification is an **SVM.** According to most sources an SVM is a slower way to classify reviews, but much more accurate than Naive Bayes.

The SVM also needed a set of features in order for it to work, and it turned out the set of features was the same bag of words Naive Bayes needed; however, it did need to be stored in a slightly different manner. Essentially every unique word in the data set is given an integer index in a dictionary. Then each review would be translated to be numeric representation of the review. For Example:

Word Dictionary

0: and

1: the

…

45: It

…

12343: Good

…

48572: was

...

Review 0: It was Good [Positive]

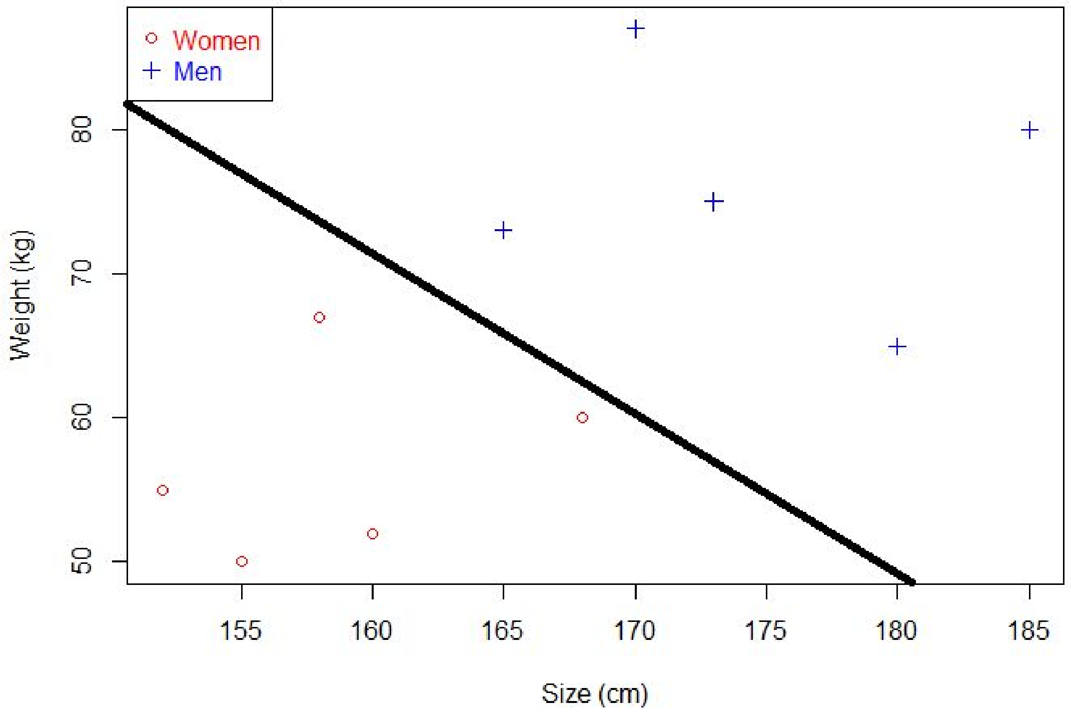
->

[0, 45][0, 48572][0, 12343]

In the tuple above the first number represents the review (document), and the second number references the id of the word. Transferring the words to numbers makes it possible for the SVM to process. The review is also labeled as positive and within the SVM will always be marked as positive, also causing the features to become associated with positive reviews. Essentially in terms of input and output SVM and Naive Bayes are exactly the same; however, the internal workings are completely different. The goal of an SVM is to

Find an optimal ***hyperplane*** *that effectively separates the different classes of training data.*

Essentially a hyperplane is just a higher dimensional version of a line. The following graph from svm-tutorial.com depicts what a 2 dimensional SVM would look like. This svm essentially tries to determine if a subject is a man or woman based on weight and height.



The dots and crosses in the graph represent if the subject is a man or woman, and their locations are based on the weight and height. The line in this example is able find a place, that makes it so on one side of the line men are present, and on the other women are present. Of course in this example there are many more features that could be used to classify these two classes, such as hair length, foot size, etc. For this reason an SVM is not limited to two dimensions, or three dimensions. An SVM can be used to represent thousands of dimensions, and in text classification it does. The most frequent words in the dictionary are the features of the SVM in text classification, so instead of using weight and size, the frequency of words such as good, or fantastic is used. This of course creates thousands upon thousands of dimensions, and if only the most frequent 10,000 - 20,000 words are selected this should cause no problem for the SVM.

After all the data has been parsed and put into the SVM, the hyperplane must be created. In the same way the equation for a line is y = ax + b, the equation for a hyperplane is:

$$\mathbf{w}\hat{}\,T\cdot \mathbf{x}=0$$

W and T in the equation are both vectors and in two dimensions it is proven that **w^*T*•x** = y - ax - b, further proving a hyperplane is nothing more than a separating line in a multidimensional space. In order to find an optimal hyperplane two hyperplanes need to be made that separate the data with no points between them, and from there the hyperplanes need to be moved so that their distance is maximized. In order for the equation to always match y - ax - b, we need to make our formula w^T • x + b, the reason it does not need to be mentioned is because **w^*T*•x** + b = y - ax - b will cause the b's to be cancelled out making the true equation for the hyperplane not need the constant..

To further create the hyperplane in the SVM we need three hyperplanes described by the equations below.

$$\mathbf{H_0} = \mathbf{w}\hat{}\,T\cdot \mathbf{x} + \mathbf{b} = 0$$

$$\mathbf{H_1} = \mathbf{w}\hat{}\,T\cdot \mathbf{x} + \mathbf{b} = 1$$

$$\mathbf{H_2} = \mathbf{w}\hat{}\,T\cdot \mathbf{x} + \mathbf{b} = -1$$

This will cause H_0 to be equidistant from the margins. We also need to set up constraints for the following hyperplanes that ensure that accurate separation of the classes occur.

$$\mathrm{w}\hat{}\,T\cdot \mathbf{x_i} + \mathbf{b} \geq \mathbf{1} \textbf{ for } \mathbf{x_i} \textbf{ having the class 1}$$

$$\mathrm{w}\hat{}\,T\cdot \mathbf{x_i} + \mathbf{b} \leq \mathbf{1} \textbf{ for } \mathbf{x_i} \textbf{ having the class -1}$$

These constraints must be respected otherwise the SVM results will not be accurate. Once H_1 and H_2 meet the constraints the hyperplanes are correct and can actually be used to start classifying data. The formal mathematics behind the constraints is expressed more concisely, but less clear by the final definition found after combining the two equations.

$$\mathbf{y_i(w^{\wedge}\mathit{T} \cdot x_i + b) \geq 1 \text{ for all } 1 \leq i \leq n}$$

Of course though this only creates a hyperplane that satisfies the constraints and not one that maximizes the distance between the two hyperplanes. The best SVM will be able to distinguish between classes as clearly as possible and this can only occur once all hyperplanes are maximally separated. Using H_1 and H_2 we create a variable m that is the perpendicular distance between both hyperplanes. We also will create a variable x_0 that is the point m starts at on the hyperplane H_1. We want to get the point on H_2 that m will reach from x_0, but that is impossible as m is simply a scalar. In order for m to become a vector value we need to gather the direction of m and create a vector that is perpendicular to H_2 with magnitude m. We know a vector that is perpendicular to H_2 which is of course it's w value from the equation of the hyperplane. With this we create vector k to satisfy what we need and get the following equation.

$$\mathbf{k=mu=mw/ /\!/ w /\!/}$$

K is now a vector between the two hyperplanes, making it possible to find a point on the other hyperplane which will be referred to as z_0. We find this by simply adding the point x_0 and k. Using all the data we now have it is possible to actually calculate the margin of the hyperplanes as we can use (x_0 + k) instead of z_0 in the equation for hyperplane 2. Further reduction gives us the equation of the margin, which is:

$$\mathbf{m=2/ /\!/ w /\!/}$$

This equation will now allow us to find and compare the different margins between potential hyperplanes. The only option we have is to change w and as w increases m will always decrease. Our goal is to maximize the margin within the constraints listed above; thus the final step is to simply minimize the w subject to the constraints. Once the minimum m is found the classification stage ends and a document can be placed in the SVM. Depending on which side of the hyperplanes the document lands in it is placed in either class A or B.

Implementation

The implementation for the project currently relies heavily on Scikit. The first thing i did was simply see what the most common words were in the data set, and how to effectively remove all the stopwords (“and”, “the”, “as”, etc.). After this was done i observed different combinations of feature sets. For example i would filter out the reviews people did not find helpful, or i would use the summary of the reviews provided rather than the actual text of the reviews. This also makes it a lot faster as there are not many stop words, there are less words to process, and many unhelpful reviews are omitted. The process of going through all 82 million reviews takes an incredible amount of time, so finding any way to make it more efficient is completely necessary. It is most definitely doable, but parsing all 82 million reviews can take quite some time.

When it came to actually building the SVM I heavily relied on scikit. Scikit is an incredibly powerful API that includes many different machine learning algorithms. The first thing i did when using scikit was appended all the reviews into a list, and created another list stating which class the review belonged in. I then passed the list into a vectorizer which created the numeric representation of the reviews mentioned in the research section of the report. The numeric representation was then paired with a class ‘negative’ or ‘positive’. Next, the SVM was created using scikit, and then the hyperplane was also created. The program then saves the SVM, and Vectorizer so that it will not have to be recomputed every time one wishes to predict a review. In a separate program, the SVM and vectorizer is loaded and from there placed into the SVM, which observes which side of the hyperplane it falls into and usually correctly predicts if the review is positive or negative.

A few roadblocks i ran into, was that there are many more positive reviews in the dataset. This caused just about all the reviews to be seen as positive as there was little evidence for a review being negative, but abundant reason to see one as positive. To fix this i made sure that an equal amount of positive and negative review were analyzed.

Once the basic SVM was created the next step was to see if it was possible to give reviews actual scores, so instead of simply giving negative or positive results, give output such as 1.0, 2.0, 3.0, 4.0, or 5.0. The SVM approach remained largely the same, just now the machine had to make multiple hyperplanes for the multiple classes. This was successfully achieved by making the five classes, then instead of matching the reviews with 'negative' or 'positive' I would simply give it a '1.0, 2.0, 3.0, 4.0, or 5.0' instead. This method seemed to be much less accurate as there is more room for error; more information can be found in the result analysis section.

The next step of implementation was to create a Naive Bayes implementation. This version uses the same parser and vectorizer to go through the Amazon Review Dataset, but a different method to actually classify the reviews. The actual implementation uses the Multinomaial Naive Bayes method. In the case of Naive Bayes the classification steps merely connects the training data with the classes assigned to it, making it significantly more simple than the SVM, which has to calculate its hyperplane and does the bulk of its work in this part. More information can be found on multinomial naive bayes within the research section.

In order to time each step of both the classification and prediction the program utilizes python's time library. The method used in the clock method, which contains the current processor time instead of just the regular time. This makes it so the timing won't be thrown off by other processes running, or any other outside effects. In order to calculate the accuracy reviews that were not included within the classification stage are inputted to the saved classifier object. If the review is in the correct category it is recorded and at the end of processing is divided by the total.

A benchmark tester was also created to run the programs and output their data to a csv. Excel is then used to make the many graphs found in the results and analysis portion. In addition testing software was made to generate the reviews that were mispredicted and to check if the results are better if the summary of the review, or the actual review data, is used.

Additional things done in terms of implementation was creating something similar to the vectorizer from scratch. Essentially the first thing i did in the project was create a program that found the words more associated with negative reviews, and words more associated with positive reviews. This was done before an actual machine learning technique was chosen as a way to see what the review set looked like. It was found that the most common negative words were don’t, buy, poor, waste, bad, money, work, product, quality, and disappointed. Then the most common positive words were great, good, book, best, love, excellent, read, one, works, and nice. This program filtered out the most common stopwords, and only picked unique positive words. This portion is not used in the final result, but is a good way for a human to look at the words more frequently used in positive and negative reviews.

Results & Analysis

The largest goal of the project was to analyse the results of the machine learning algorithms and successfully determine which classification methods were best for analyzing the reviews. Before implementing I fully expected that the SVM would be much more slow, but more accurate; though, the Naive Bayes approach would be much faster, while being less accurate. The most successful classification algorithm would be one that can classify, and predict reviews very fast, while also upholding very accurate results. This report does not include the times taken to actually parse the review set as this would change based on what review set a user has, and how that set is formatted. In addition, the parsing processes will be the same for both the Naive Bayes and SVM implementation. Parsing is by far the most intensive part of the algorithm though, considering multiple gigabytes are being processed, and in my case on a very weak virtual machine.

All the tests include the following attributes:

- A negative review is one with a score of 2 or lower.
- A positive review is one with a score higher than 2.
- The times were calculated with time.clock()
- Ran on a Linux Ubuntu Virtual Machine with 10,998 MB of base memory.

The following small tests contain the following attributes:

- Reviews processed went from 10,000 reviews to 90,000 reviews.
- Only reviews that had a helpful rank of 4 or above were included.
- The review summary was parsed rather than the actual review text.
- Predictions were ran over a set of 5,000 reviews not included in the classification step. Prediction times refer to time taken to predict all 5,000.

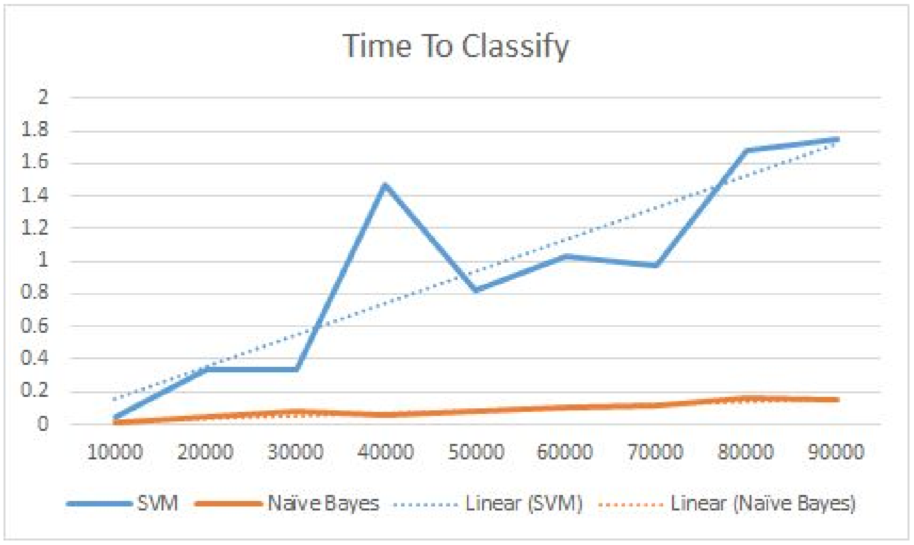


Figure 1

In figure 1 it is clear that the SVM and Naive Bayes have about the same run time to classify reviews at the beginning, but they quickly separate. It is hard to see with only ten inputs if the growth in time is linear, but it is clear the SVM begins to get much slower after only about 90,000 reviews, which in terms of getting the most accurate results is not many reviews. Naive Bayes seems to be linear, but as with SVM this test is a little too small. Accuracy is one of the most important aspects of the results and as seen in figure 2 SVM's completely outperform Naive Bayes. The margin of difference is not that large, but the tradeoff when it comes to time seems to not be that large.

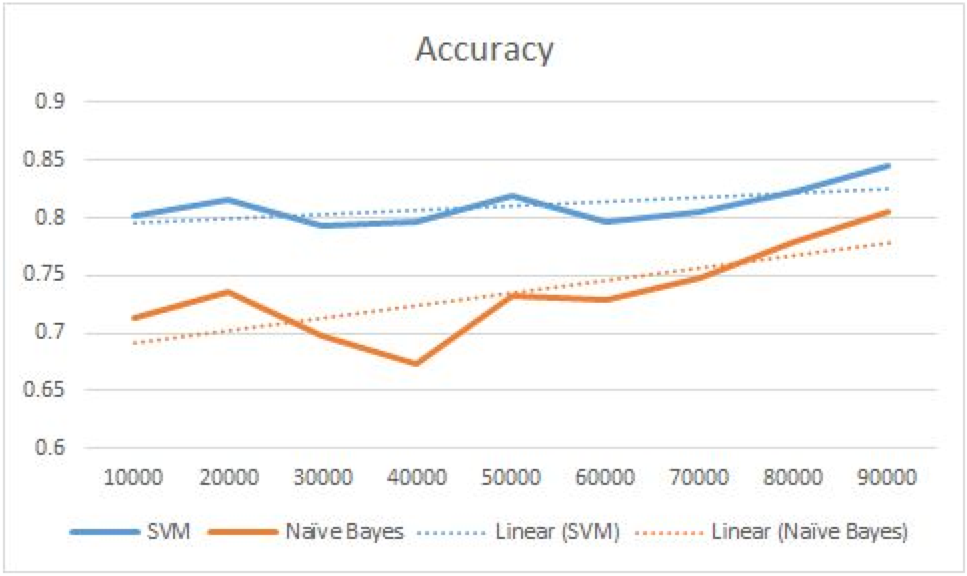


Figure 2

Waiting for just about two seconds does not make a significant difference especially as there is an improvement of about .5 percent; however, both the SVM and Naive Bayes can perform better than this. When given a larger set the machines should be much more accurate, but also slower. One important aspect to observe is that the accuracy does not always go up when given more reviews. This is likely because one set of 10,000 will have more false positives than other sets. The SVM and Naive Bayes are both impacted by these false positives as the accuracy moves in a similar pattern, but it seems that naive bayes performs worse when exposed to these false positives in the 20,000 to 40,000 range. The SVM goes from .816 to .7966 in the 20,000 to 40,000 range while naive bayes goes from .7362 to .6728.

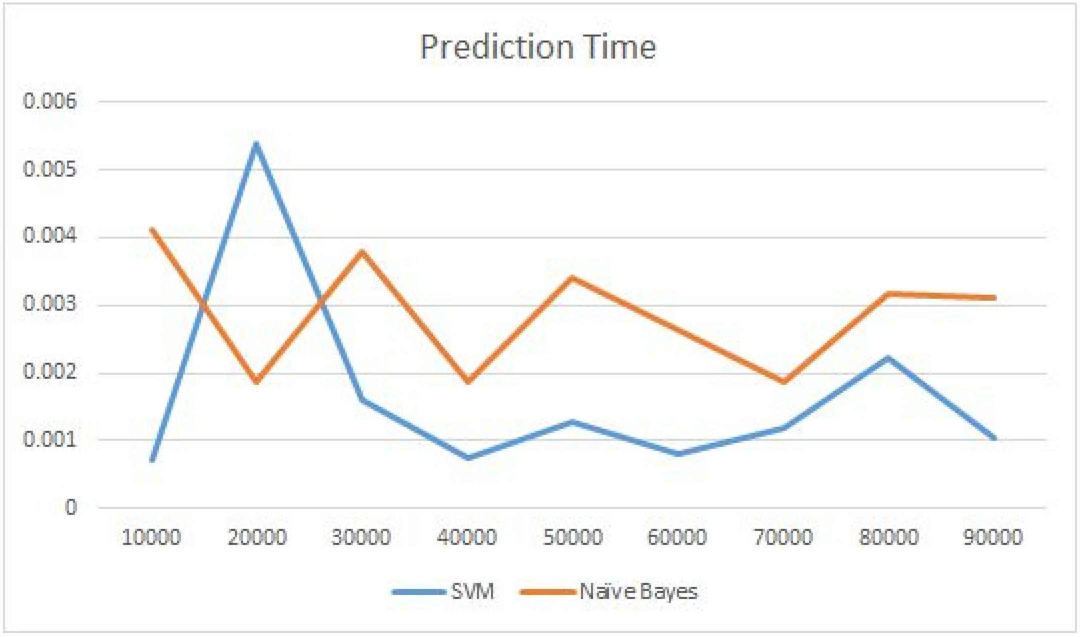


Figure 3

The prediction time is much different than one would expect. The SVM seems to be much lower when it comes to simply prediction. These predictions are formed after the classification has been performed and all the reviews have been fit for predictions. With this small data set it is difficult to make conclusions, but it seems as if SVM's are much faster at predicting. The results are so fast and close together that a bigger data set will be able to make better conclusions.

<u>The following medium tests contain the following attributes:</u>

- Reviews processed went from 100,000 reviews to 1,000,000 reviews.
- Only reviews that had a helpful rank of 4 or above were included.
- The review summary was parsed rather than the actual review text.
- Predictions were ran over a set of 5,000 reviews not included in the classification step.
- Prediction times refer to time taken to predict all 5,000.

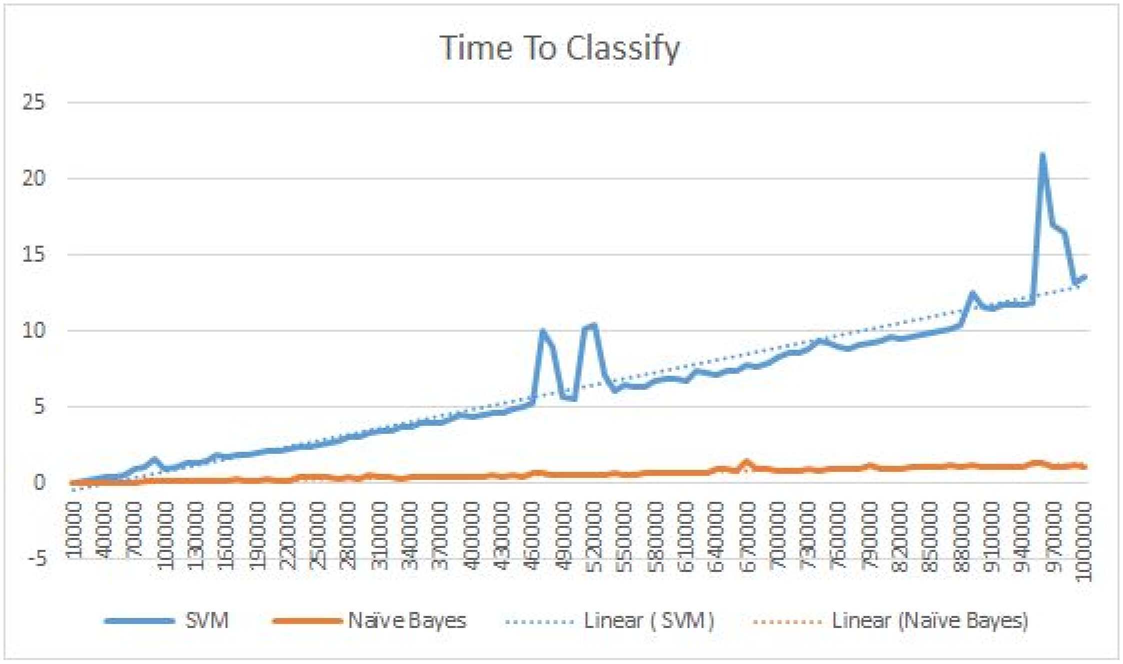


Figure 4

The time to classify in figure 4 continues to run as expected on the larger test, but the difference between the classification time for an SVM and Naive bayes continues to grow. SVM time seems to always be getting larger, while Naive Bayes time seems just about constant. Figure 5 shows the actual growth of Naive Bayes and it is clear it is still linear, but the slope remains much less than the SVM's

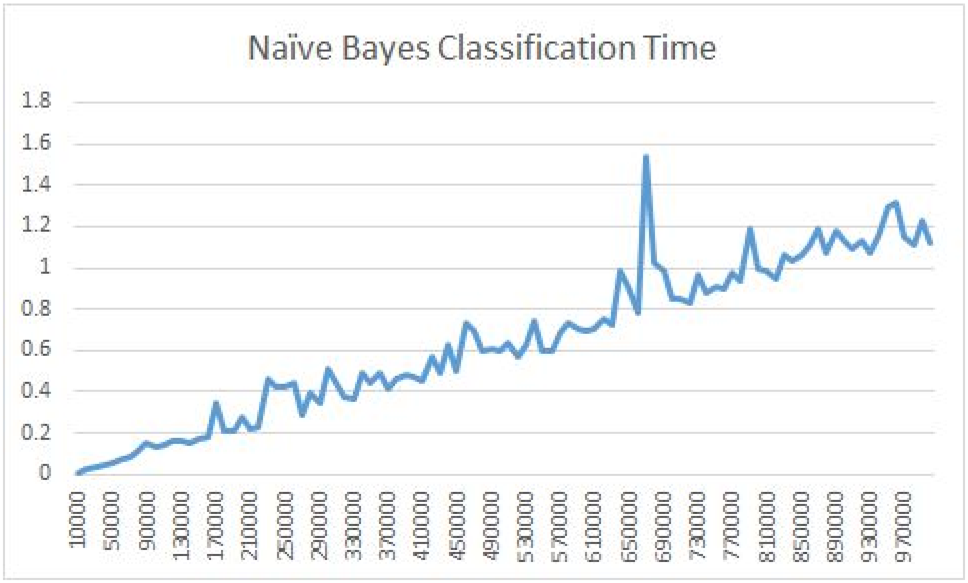


Figure 5

Accuracy receives a huge boost when applied to the much larger data set, but it seems to not grow as much once it reaches about 28,000 reviews. In fact, this is when both algorithms have their best accuracy rating. Just as before, both algorithms follow the same pattern and again, naive bayes seems to be impacted far more by false positives as it falls far below SVMs at around 73,000 reviews. Naive Bayes is never able to become more accurate than SVMs thus showing how SVMs are the more accurate algorithm.

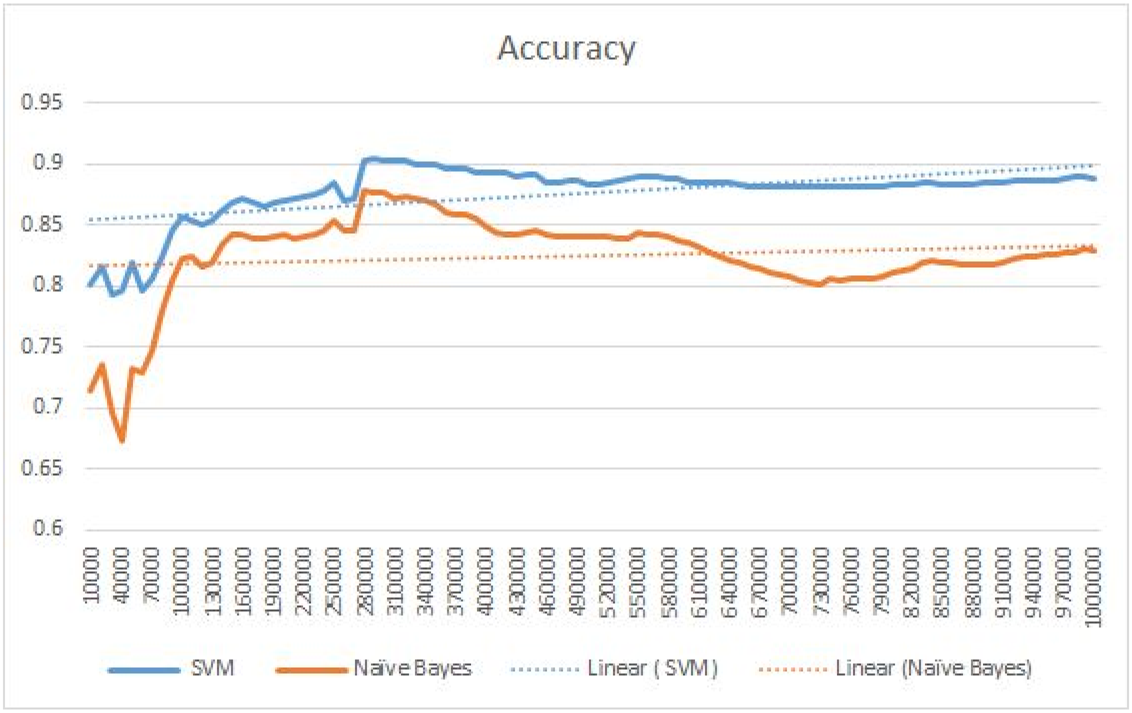


Figure 6

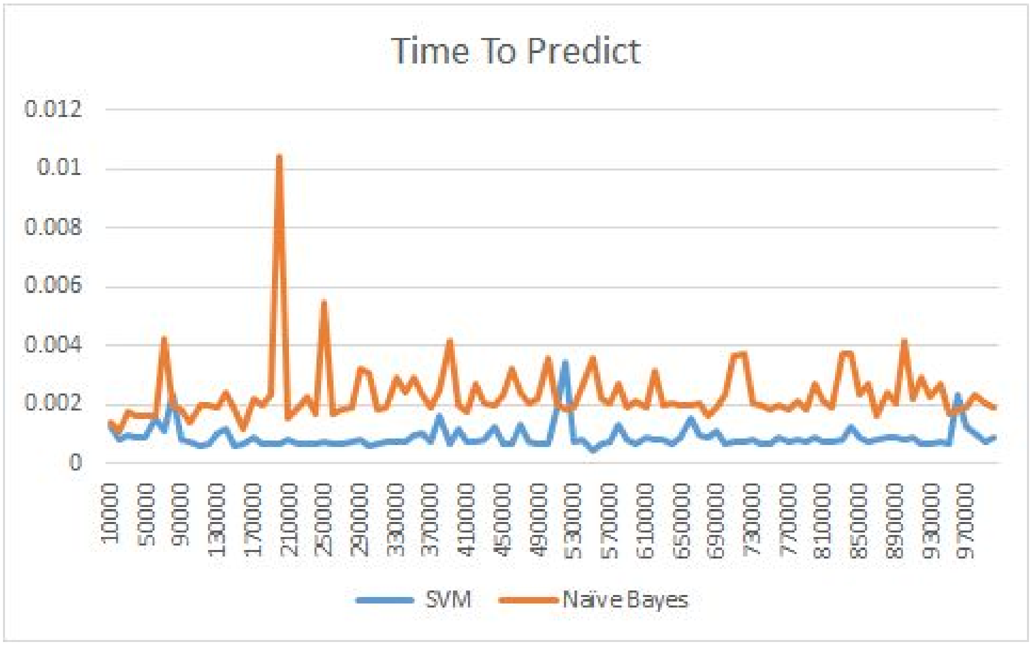


Figure 7

The prediction times remain with no growth. The SVM is a little faster than Naive Bayes, but as there is so little growth in either, any difference found is largely negligible.

When the same tests are performed on a review set from 2 million to 10 million the results remain largely the same. The only differences is that naive bayes actually does pass the SVM in accuracy as it gets to 10 million reviews. This is likely coincidental as an SVM is usually much more accurate, but it is interesting that with a huge data set the differences become negligible, yet naive bayes becomes significantly faster to classify; however, once the classifiers has been saved there is really no difference in the prediction time. The graphs showing this are figure 8, 9, & 10 provided below for your reference.

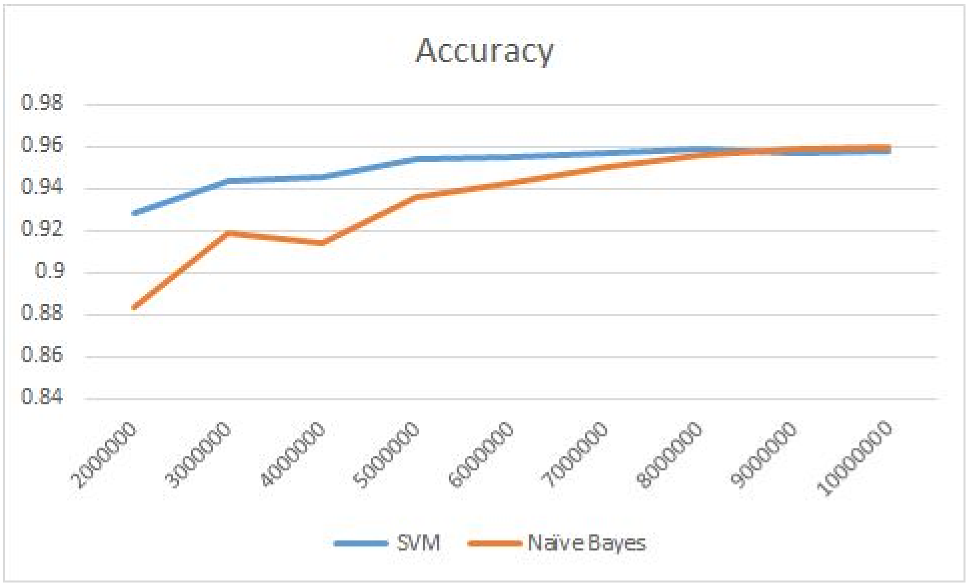


Figure 8

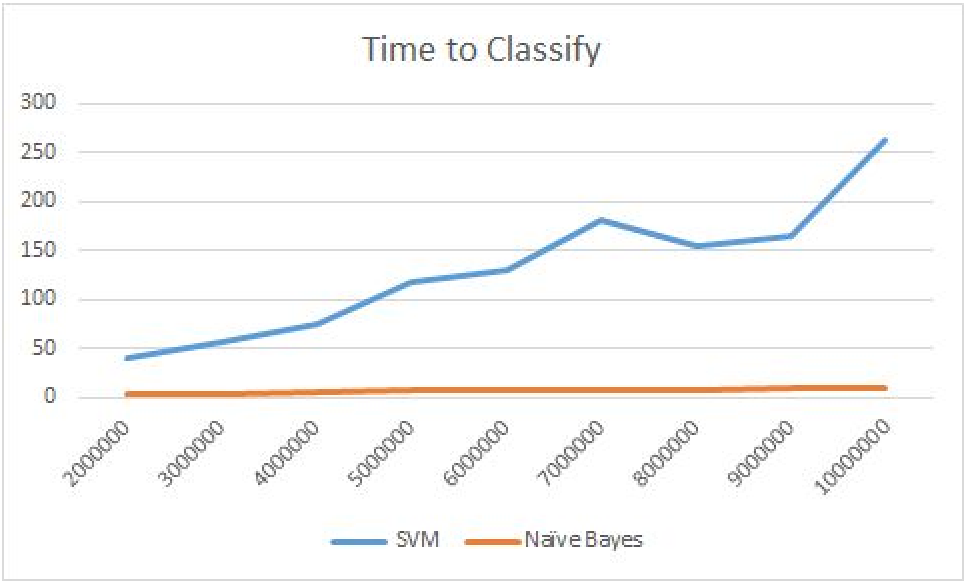


Figure 9

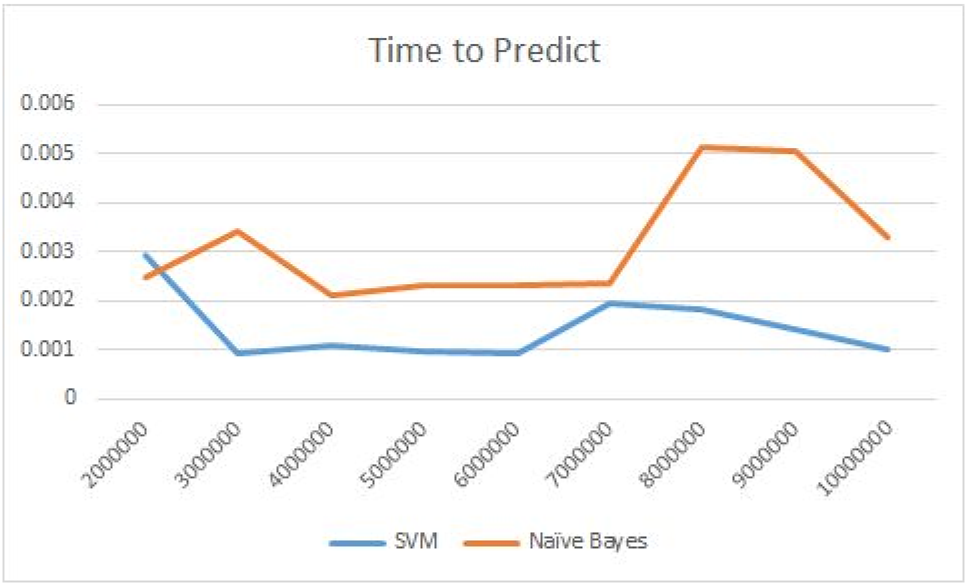


Figure 10

When it comes to review classification that assigns a score rather than just negative or positive the SVM is much less accurate. At 10,000 reviews the stats are the following:

Prediction Time: .0022 (S)

Classification Time: .236 (S)

Accuracy: .29

Near Accuracy: .66

The prediction time is just as low as the other classifiers, but the classification time is much more than the regular SVM. The regular SVM is able to classify 10,000 reviews in about .06 seconds compared to the .236 seconds here. The accuracy is also incredibly low at just .29 compared to the regular SVM, which has an accuracy of .80. This is off course partly due to the greater room for error. The regular SVM has to classify two classes giving it a more likely chance of being correct, while the new classifier has four other choices to make a mistake on. When looking at the near accuracy, which allows for the score to be one off it is much more accurate at 66 percent. Just as the other SVM, this machine is much more likely to be accurate with a higher number of reviews as seen below.

Number of Review: 1,000,000

Prediction Time: .02 (S)

Classification Time: 193.13 (S)

Accuracy: .457

Near Accuracy .784

In the regular SVM the classification time does not reach that length before 10 million reviews. Even with the poor time performance the accuracy of .78 for such a large training set is far below what is expected, as the regular SVM can get about a .88 accuracy. When tested with a set of 5,000,000 reviews it takes a whopping 1022 seconds, and only achieved an accuracy of .5306; however, the near accuracy was acceptable at 80.8 percent.

On the very accurate SVM with a prediction accuracy of around 94 percent, looking at the reviews that were misclassified shows the limitations and possibly strengths of the SVM.

Should be Positive

Prediction:Negative, Actual:3

I had high expectations for this book, and while it was as easy to read and rather fast-paced as everyone says, I just didn't like it as much as I expected. for the first 50 pages, nothing interesting happened, and characters that I didn't care about were thrown at me rather rapidly. I didn't think that the main character was fleshed out enough, and Marlena was nothing more than a blubbering female character for the entire novel.Disappointing. Rather dull, really, and way more *depressing than I was expecting. There wasn't anything redeeming about it. I don't recommend it.*

The review shown here was misclassified when taking the user score into account; though, the text of the review is very negative. This is a case where the SVM has actually seemed to give the review a better rating that the user did. There is very little reason for this review to be given a 3, and this should not be seen as a misclassification. This type of problem exists throughout the set of incorrectly classified reviews and shows that human error will always have a negative impact on the SVM. If this review was parsed by the classifier, it would also be a reason misclassifications could happen because it might make the SVM think words like depressing, disappointing, and dull are positive.

Should be Negative

Prediction:Positive, Actual:1

does anyone care that zac isn't the voice on the album and he lip sings in the movie? who is the person really singing?

In this case the review text itself does very little to actually define itself as negative. The reviewer really just asks a question, then assigns the product a low score. There is no reason to assign this review a positive or negative score. Of course reviews like this are also used to build the classifier which can cause the classifier itself to have problems. Sometimes it is hard for a human to distinguish between a positive or negative review, so it should be no surprise it is challenging for a computer.

Overall it seems that both attempts to classify text are equally good. They both achieve very similar accuracy results once exposed to a large set of reviews; however it is still shown that naive bayes is more affected by false positives. Naive Bayes seems to be the better approach when given a large set of reviews as it is faster and just as accurate. Then an SVM seems to be better when given a smaller set of reviews as it is much more accurate from an earlier stage. It is also probably near impossible for each algorithm to reach perfect accuracy as many reviews are very objective and do not directly share a positive or negative attribute.

Future Work

- Neural Networks Implementation & Comparison.
- K Nearest Neighbor Implementation & Comparison.
- Implementation of algorithms, and removal of API usage.
- Comparisons of different algorithms on different data sets.
- Using current implementations, fit classifiers to hate speech or not hate speech data set (This was the original goal of the project, but no data set of hate speech could be found).

Citations

Image-based recommendations on styles and substitutes

J. McAuley, C. Targett, J. Shi, A. van den Hengel

SIGIR, 2015

pdf

Inferring networks of substitutable and complementary products

J. McAuley, R. Pandey, J. Leskovec

Knowledge Discovery and Data Mining, 2015

Pdf

Kowalczyk, Alexandre. "SVM Tutorial." *SVM Tutorial*. N.p., 2 Nov. 2014. Web.

Appendix, Figures

Appendix, Code

```
def saveObject(object, fileName):

        with open(fileName, 'wb') as newFile:

                pickle.dump(object, newFile, -1)



def parse(path):

        g = open(path, 'r+')

        for l in g:

                yield eval(l)



def sortReviews(reviews, reviewLabels, cap):

        accept = 0

        reject = 0

        negCount = 0

        posCount = 0

        oneCount = 0

        twoCount = 0

        threeCount = 0

        fourCount = 0

        fiveCount = 0

        itr = 0



        print "Begining"
```

```
for review in parse("./UncompressedData.json"):

        if review['helpful'][1] > 3:

                if float(review['helpful'][0])/review['helpful'][1] > .50:

                        review['summary'] = review['summary'].replace(".","")

                        review['summary'] = review['summary'].replace("?", "")

                        review['summary'] = review['summary'].replace("!", "")

                        review['summary'] = review['summary'].lower()




                        if review["overall"] >= 3.0 and posCount < cap/2:

                                reviews.append(review["summary"])

                                posCount+=1

                                reviewLabels.append("Positive")

                                accept +=1


                        elif review['overall'] < 3.0:

                                reviews.append(review["summary"])

                                reviewLabels.append("Negative")

                                negCount+=1

                                accept+=1

                else:

                        reject+=1


        else:
```

```
                    reject+=1

                itr+=1


                if (accept == cap):

                    break



        def vectorize(reviews):

            print "Begin Vectorization"

            vectorizer = TfidfVectorizer(min_df=5, max_df = 0.8, sublinear_tf=True,
use_idf=True)

            print "End Vectorization"

            print vectorizer

            return vectorizer



        def runSVMClassifier(cap):

            reviews = []

            reviewLabels = []


            sortReviews(reviews, reviewLabels, cap)

            vectorizer = vectorize(reviews)

            reviewVectors = vectorizer.fit_transform(reviews)


            start = time.clock()

            classifier = svm.LinearSVC()
        classifier.fit(reviewVectors, reviewLabels)

        end = time.clock()


        classifyTime = end - start

        print "Time to classify", (end - start)

        saveObject(classifier, "classifier.pkl")

        saveObject(vectorizer, "vectorizer.pkl")


        return classifyTime

def naiveClassify(cap):

        reviews = []

        reviewLabels = []


        sortReviews(reviews, reviewLabels, cap)

        #print reviews

        #print reviewLabels

        vectorizer = vectorize(reviews)

        reviewVectors = vectorizer.fit_transform(reviews)

        print reviewVectors


        print "Create NB"

        start = time.clock()

        classifier = MultinomialNB()

        classifier.fit(reviewVectors, reviewLabels)
        end = time.clock()

        classifyTime = end - start

        print "Time taken to classify ", (end - start)

        saveObject(classifier, "naiveClassifier.pkl")
        saveObject(vectorizer, "naiveVectorizer.pkl")

        return classifyTime
def analyzeReviews(classifier, vectorizer, cap):
        totalCount = 0
        correct = 0
        itr = 0
        test = []
        score = []
        offset = 30000000
        for review in parse("./UncompressedData.json"):
                if itr < offset:
                        itr+=1
                else:
                        itr+=1
                        test.append(review['reviewText'])
                        score.append(int(review['overall']))
                        if itr == cap + offset:

                                break

                #if (itr % 10000) == 0:

                #       print itr



        testVectors = vectorizer.transform(test)



        start = time.clock()

        prediction = classifier.predict(testVectors)

        end = time.clock()

        print "Time to predict ", (end - start)

        f = open("incorrectlyClassifiedReviews.txt", "w")



        itr = 0

        for i in score:

                totalCount+=1

                if i >= 3.0:

                        if prediction[itr] == "Positive":

                                correct +=1

                        else:

                                f.write("Should be Positive\n Prediction:" +
str(prediction[itr]) + ", Actual:" + str(i) + "\n" + test[itr] + "\n\n")

                if i < 3.0:

                        if prediction[itr] == "Negative":
```

```
                                        correct+=1

                                else:

                                        f.write("Should be Negative\n Prediction:" +
str(prediction[itr]) + ", Actual:" + str(i) + "\n" + test[itr] + "\n\n")

                        itr+=1



                print float(correct) / totalCount

                return (end - start), (float(correct)/totalCount)



        def enterReview(classifierFileName, vectorizerFileName, numReviewsToPredict):

                classifier = pickle.load(open(classifierFileName, "rb"))

                vectorizer = pickle.load(open(vectorizerFileName, "rb"))

                return analyzeReviews(classifier, vectorizer,numReviewsToPredict)



        class Benchmark():

                def __init__(self, numReviews, timeToPredict, timeToClassify, accuracy,
typeOfClassifier):

                        self.numReviews = str(numReviews)

                        self.timeToPredict = str(timeToPredict)

                        self.timeToClassify = str(timeToClassify)

                        self.accuracy = str(accuracy)

                        self.typeOfClassifier = typeOfClassifier

        f = open("benchmarks_large.csv", 'w')

        f.write("NumReviews, TimeToPredict, timeToClassify, Accuracy, type\n")
        i = 1000000

        benchmarks = []


        while i < 11000000:

                timeToClassify = runSVMClassifier(i)

                timeToPredict, accuracy = enterReview("Classifier.pkl", "Vectorizer.pkl", 5000)


                benchmarks.append(Benchmark(i, timeToPredict, timeToClassify, accuracy, "SVM"))

                i = i + 1000000


        i = 1000000


        while i < 11000000:

                timeToClassify = naiveClassify(i)

                timeToPredict, accuracy = enterReview("naiveClassifier.pkl",
"naiveVectorizer.pkl", 5000)


                benchmarks.append(Benchmark(i, timeToPredict, timeToClassify, accuracy,
"naive"))

                i = i + 1000000


        for b in benchmarks:

                f.write(b.numReviews + ", " + b.timeToPredict + ", " + b.timeToClassify + ", " +
b.accuracy + ", " + b.typeOfClassifier + "\n")
```

```
def printDict(dictionary, minCount, compDict, wordCount, compWordCount):

        itr = 0

        retList = []


        while dictionary and itr < minCount:

                result = max(dictionary, key=lambda i: dictionary[i])

                print result, dictionary[result]


                temp = dictionary[result]

                del dictionary[result]


                if(compDict.has_key(result)):

                        if float(compDict[result])/compWordCount > float(temp)/wordCount:

                                print "Reject String"


                        else:

                                print "Accept String"

                                itr+=1

                                retList.append(result)


                else:

                        print "Accept String"

                        itr+=1

                        retList.append(result)
        return retList


def writeToFile(path, list):

        file = open(path, "w")


        for i in list:

                file.write(i + "\n")

        file.close()


posWordCount = 0

posWordDict = {}

negWordCount = 0

negWordDict = {}

itr = 0


stop = stopwords.words('english')


print 'Begining'


for review in parse("./item_dedup.json.gz"):

        review['reviewText'] = review['reviewText'].replace(".","")

        review['reviewText'] = review['reviewText'].replace("?", "")

        review['reviewText'] = review['reviewText'].replace("!", "")

        review['reviewText'] = review['reviewText'].lower()
```

```
words = review['reviewText'].split()


if review['overall'] >= 3.0:

        for i in words:

                if i in stop:

                        continue

                #print i

                posWordCount += 1

                if i in posWordDict:

                        posWordDict[i]+=1

                else:

                        posWordDict[i] = 1


else:

        for i in words:

                if i in stop:

                        continue

                #print i

                negWordCount += 1

                if i in negWordDict:

                        negWordDict[i]+=1

                else:

                        negWordDict[i] = 1
        itr+=1


        if (itr % 10000) == 0:

                print itr

        #if (itr % 10000) == 0:

        #       break


print "Positive Words"

tempPosWordDict = posWordDict.copy()


posList = printDict(posWordDict, 100, negWordDict, posWordCount, negWordCount)


print "Negative Words"

negList = printDict(negWordDict, 100, tempPosWordDict, negWordCount, posWordCount)


print "Positive"

print posList


writeToFile("PositiveWords.txt", posList)


print "Negative"

print negList
```

```
        writeToFile("NegativeWords.txt", negList)


        print "PosWordCount is: ", posWordCount

        print "NegWordCount is: ", negWordCount
```

* When it came to classifying by score the only difference was that there were five classes and more comparisons to see what class a review belonged to.

* To run the program ensure you have python, pickle, and the amazon data set included if you wish to classify. If you only want to predict, running python inputSingleReview.py "<Review here>" will predict the result of one review using the SVM. Running with -n will do it through naive bayes, and -s will do it by score (please put flag before review and only use one flag per run).