Sarcasm Analysis On Steam Reviews

Matthew Rinker
Denison University
Granville, OH, USA

ABSTRACT

When considering a purchase one often looks to the experiences of others to provide guidance on the quality of a product. Video games are no different. Gamers find themselves spending from tens to hundreds and in some cases even thousands of hours playing their favorite games. However, quite often these games are expensive purchases. Steam is a large online retailer, run by the ValvE Corporation, which offers digital downloads of over fifty thousand titles. On this platform Steam has given their users the ability to write and post reviews without any official oversight. The video game community is known for their sometimes abrasive humor and tendencies to enjoy leading others on through falsehoods. In this paper I will explore Steam reviews through the lenses of Sentiment Analysis, Sentance Structure Analysis, and through examination of excessive use of punctuation and capital letters to identify fake, false, and sarcastic reviews.

1. INTRODUCTION

ValvE Corporation provides the world's largest online retailer of video games, Steam. As of November of 2018 Steam boasts a catalogue of 53,880 products ranging from Video Games, Downloaded-able Content from Video Games, and various other pieces of software. The number of Video Games alone that Steam offers is an astronomical 28,391 games. Naturally, it is impossible for any user to try even a small sampling of those products. As such, decisions must be made of what to purchase. ValvE implemented a user review system to aid in this process.

Steam Users may submit reviews for any game which they own. These reviews are classified by the reviewer as either Recommended or Not Recommended. These reviews are then further classified by other users as either Helpful or Not Helpful. In general, this is a useful tool to weed out fake reviews. However, the gaming community is somewhat well known for their penchant for jokes and enjoyment of messing with others. Due to this fake and humorous reviews often find themselves topping the lists above their genuine counterparts.

This is an unfortunate occurrence that is simply a byproduct of the crowd sourced nature of ValvE's review system. While allowing users to review and essentially rank the products on their platform for them they dramatically reduce costs but sacrifice a level of integrity and quality in their reviews. However, suppose there was a way to automatically weed out the fake, sarcastic, and mislabeled reviews. This would significantly reduce the drawbacks of this automatic system.

In this paper I attempt to develop a model which will identify sarcastic reviews on the Steam platform. This will be a difficult task due to the nature of sarcasm. In speech, sarcasm identification is predicated on the inflection and tone of the speaker. In text, these clues are not apparent. I will be using sentiment analysis to detect commonly negative or sarcastic words. I will also attempt to use this to help tag reviews which contain hyperbole. I will also analyze parts of speech to compare sentence structure to see if there is a pattern between certain sentence structures and sarcasm. Finally, I will use punctuation and capitalization analysis to detect excessive punctuation and capitalization which are more common in sarcastic reviews.

2. RELATED WORK

There have been a few notable projects on the analysis of game reviews on the Steam platform in recent years. These projects primarily focused on the sentiment analysis and using various sentiment analysis algorithms to identify the class of a review.

One such project was done by Standford students and primarily featured the Naive Bayes algorithm, Linear Regression Models, a Lexicon Score algorithm, and a modified form of Turney's algorithm. [1] This project focused entirely on which algorithm produces the most accurate classification. Their results found the Naive Bayes algorithm to be moderately accurate. This could be problematic for my results, however their results find that the Naive Bayes algorithm is reasonably accurate. I also believe that since they neglected to account for intentionally false and sarcastic reviews that their results may be skewed. One novel aspect they introduced was noise reduction in the review dataset. An interesting issue arising from the use of a crowd sourced dataset on an online platform is the introduction of online slang, and shorthand in the reviews. By employing noise reduction they sought to eliminate those factors from their calculations. While this is a good idea, I disagree with

their implementation. They used a nltk function to remove misspelled words. However, I find that in conjunction with this it may help to implement into the training vocabulary tagged versions of common slang and shorthand to enhance the efficiency of the classification algorithms.

Another project was conducted by a pair of Polish researchers [3]. Their paper also focused on sentiment analysis, however their approach differed and by using some of their techniques it could lead to greater insight into my problem. They use the usefulness measure as voted by the community as well as a funny score to help guide their calculations. I believe that by factoring in the funny score of reviews it may help identify sarcastic reviews as often the community finds these to be humorous. Additionally, the motivation behind writing sarcastic reviews is most commonly to be humorous and to acquire more helpfulness votes. Thus, by analyzing these metrics it may assist me in classifying sarcastic reviews.

The last significant project which I came across was written as a blog post detailing a personal project of data scientist, Scott Burger [2]. His project mainly focused on emotion classification and is the closest to what I want to explore. His techniques could improve my sarcasm analysis as I could use it to strengthen my hyperbolic, capitalization, and punctuation analysis. By removing those reviews with excessive hyperbole, capitalization and punctuation analysis that his classifier marks as having angry emotion markers I can further refine my subset and see if my method has any merit in terms of sarcasm classification.

3. METHODS

Steam is a fantastic resource for analysis due to the sheer number of games and the different communities each game attracts. As such, I will be dividing reviews by game and looking at the ratio of sarcastic and fake reviews to real reviews.

I will use a threefold approach to discover if I can create a working model to detect sarcasm in text. I will combine use of sentiment analysis, part of speech classification, and analysis of punctuation and capitalization to attempt to identify patterns in sarcastic text. I will also be conducting some initial revisions to my training sets to better refine my results. I will be taking the concept of noise reduction from the Stanford paper [1] and will refine it to help tag slang and common shorthand to avoid these from being thrown out due to being unknowns are viewed as misspellings. An additional test I will try will be filtering by high helpfulness and humor scores and seeing if that correlates with sarcastic reviews.

3.1 Sentiment Classification

The first method I will be using is sentiment classification. Sentiment classification entails using a training set to tag commonly used words and associate them with given positive and negative tags. I will first test my classifiers using two separate tests with two different training corpuses. First, due to their length and similarity in language to video game reviews I will use the Brown Movie Review corpus. Second, I will use a subset of curated steam reviews as a training set. I will use these to analyze and pick out possible false positive reviews, that is reviews that are marked as positive but appear negative. I will then investigate these reviews, and mark by hand the ones I deem to be sarcastic in nature. In this way I hope to build a training corpus of sarcasm which can be used to identify sarcastic reviews from a set of steam reviews.

Bayes Classifiers

My primary tool for sentiment analysis will be a Bayes classifier. I will be using to versions of the Bayes classifier and ultimately going with whichever works best. I will be using my own implementation of the Naive Bayes algorithm using logarithmic probabilities and add one smoothing as well as the Bayes classifier included in the nltk module included in the python language.

Algorithm 1 The Naive Bayes Algorithm

Input: A Set of tagged Training Documents S, A set of class Tags C, A set of tagged Testing Documents T **Output**: Precision, Recall, A Set of False Positives P, A Set of False Negatives N

for c in C **do**
  Calculate number of Documents N_d
  Calculate number of Documents of class c N_{d_c}
  logprior[c] = $log(\frac{N_{d_c}}{N_d})$
  Record the vocabulary of the documents V
end for
for $w \in V$ **do**
  count occurrences of word in documents of class c $count(w, c)$
  calculate the likelihood of each word appearing in a document of class c given the previous word w' $log(\frac{count(w,c)+1}{\Sigma_{w' \in V}(count*w',c)+1})$
end for
  #End Training
for $c \in C$ **do**
  sum[c] = logprior[c]
  for $t \in T$ **do**
    for $w \in t$ **do**
      if $w \in V$ **then**
        sum[c] = sum[c] + loglikelihood[word,c]
      end if
    end for
  end for
end for
$Argmax(sum)$ to decide which class the document is
Increment True positive, False Positive, False Negative, or True negative based on the document's tagged class
if False Positive **then**
  Add to P
end if
if False Negative **then**
  add to N
end if
calculate Precision $\frac{\text{True Positives}}{\text{True Positives+False Negatives}}$
calculate Recall $\frac{\text{True Positives}}{\text{True Positives+False Negatives}}$
  return Precison, Recall, P,N

(See Appendix A for my implementation of the Naive Bayes Algorithm) This will allow me to sort out all true positive and negative results and work with the anomaly reviews. I will then train my modules using these false positive and false negative reviews after classifying them by hand to

be either sarcastic or not sarcastic. I will then use my Bayes Classifiers again to work on a new set of false positive and negatives to see if I can identify sarcastic reviews in this manner. Due to the nature of the Steam platform it will be almost impossible for me to run out of reviews even while only using false positive and false negative tagged reviews, thus the size of my corpus should not be a problem.

3.2 Part of Speech Classification

My secondary exploration will be in the sentence structure of sarcastic reviews. I will again be using reviews given as false positives and false negatives from my Bayes Classifiers. I will be using a Viterbi classifier as well as the nltk part of speech tagger to identify the sentence structure of my reviews which are tagged as sarcasm. I will then identify if there are any patterns between these reviews. Should I find a pattern I will then use my Viterbi classifier (See Appendix C for my implementation of the Vitterbi Algorithm) and the nltk part of speech tagger on a training set to attempt to identify more sarcastic reviews.

Viterbi Classifier

Algorithm 2 The Viterbi Algorithm

Input: Set of Observations O, state graph S, length of O T, length of state-graph N **Output**: best-path, path-prob

Create path matrix of size [N,T], *viterbi*
for state $s \in S$ in range $1, N$ **do**
  $viterbi[s,1] = \pi_s * \beta_s(o_1)$
  $backpointer[s,1] = 0$
end for
for time step t in range $2, T$ **do**
  for each state s in range $1, N$ **do**
    $viterbi[s,t] = max_{s'=1}^{N} viterbi[s', t-1] * a_{s',s} * \beta_s(o_t)$
    $backpointer[s,t] = argmax_{s=1}^{N} viterbi[s', t-1] * a_{s',s} * \beta_s(o_1)$
  end for
end for
$bestpathprob = max_{s=1}^{N} viterbi[s,T]$
$bestpathpointer = argmax_{s=1}^{N} viterbi[s,T]$
best-path is the path starting at *bestpathpointer* that follows *backpointer* through the matrix return best-pathm, path-prob

The Viterbi algorithm will allow me to identify the sentance structure in a bigrams model. This will give me the ability to see an abstract representation of the reviews at a language level. After running Viterbi on a set of set of reviews tagged by me as sarcastic I will pull out the bigram representations of what I view sarcastic sentences. I will then use this representation to compare and contrast other sarcastic sentences and attempt to create a template for a sarcastic sentence on the part of speech level. I will then use this to identify other reviews which contain sentences which match the sarcastic template.

3.3 Punctuation and Capitalization Analysis

Finally, I will conduct a simple analysis of excessive use of capitalization and punctuation using the sarcasm tagged reviews. I will attempt to identify if there is any correlation between excessive use of punctuation and capitalization by identifying a set of reviews with excessive punctuation and capitalization and comparing that to the sarcasm tagged reviews from my Bayes classifier results. Additionally, I hope to use Scott Burger's emotion classifying method to further weed out reviews that exhibit emotions that I believe will skew my data [2]. I also hope to try to see if taking out reviews exhibiting angry emotions and possibly looking only at those that exhibit a humorous emotion will provide a different result than on the unchanged dataset.

4. CHALLENGES

To complete this project I was met with many different challenges. These challenges ranged from the time frame of the project, to my workload in this semester, and finally to the nature of the dataset itself. All in all, there are some tactics that I wish to further investigate which I will lay out in section 7.1. Additionally, my project is focused on the identification of sarcasm. Sarcasm in speech is heavily reliant on tone and delivery, both of which are hard to translate into text. I planned to combat this specific challenge by analyzing capitalization and punctuation patterns to help determine emphasis and pauses in delivery. These would most likely be characterized by capitalization and utilization of punctuation structures, such as the ellipses, respectively.

4.1 Dataset

The dataset I used was a set of 90,000 steam reviews over 6 games. These games were: "Batman: Arkham Knight," "Counter-Strike: Global Offensive," "Dark Souls 2," "Dota 2," "Payday 2," and "Planetside 2." This specific dataset was scraped from the steam store page and was provided to me by Scott Burger [2]. However, this dataset proved to be challenging for a number of reasons.

First, these reviews were crowd sourced from a community of individuals known for their abrasive sense of humor and penchant for finding humour in the confusion and frustrations of others. To make matters worse, steam provides a "Badge" for users' profile rank for reviewing a single game. This led to a lot of nonsensical reviews on games on reviews which purely stated that they were merely reviewing the game for the badge. Additionally, there is no spelling or grammar checking associated with these reviews, this led to interesting phenomena such as reviews containing no punctuation, or no spaces. Furthermore, slang and misspellings were common. These factors of the dataset combined with the encoding of Steam's reviews itself proved to be the biggest issue. Steam's profanity filter seemed to replace profanity with a seemingly random string on unusual characters. This meant that for any reviews to be used they must first be cleaned. This concept was introduced in the stanford paper when they used their "noise reduction" algorithm to clean the dataset [1]. I, however, decided that the most accurate way to guarantee a clean dataset would be to do it by hand. This presented my most substantial challenge due to the size of the dataset and the magnitude of the undertaking. This also took up the most time out of any part of this project and led to my inability to accomplish all the different forms of analysis that I wished to take into account. Again, I will touch further upon this in section 7.1.

5. EVALUATION

I carried out the methods above and was met with mixed results. This was not entirely unexpected, however this was aggravated due to the many challenges with this project.

5.1 Procedure

To begin my analysis I started with my Naive Bayes Classifier (see Appendix A.). I trained the classifier with a simple binary classifier provided by the Brown Corpus. I decided that the best way to eliminate all inconsequential reviews (e.g. those unlikely to contain sarcasm) I must eliminate as many of the purely serious and normal reviews as possible. To accomplish this I highlighted the similarities between a standard game review and a movie review. To this end, I used my Naive Bayes classifier to sort all reviews by either positive or negative as classified by the Brown corpus. I then took all the false negative and false positive reviews to examine.

Upon inspection of my resultant dataset I found that the vast majority of sarcastic reviews in this dataset were tagged as positive. This stood to reason since not many people pretend to write a bad review for a game they actually like, people instead tend to write reviews tagged as positive which are instead disguised scathing reviews for games since most review display platforms show the positively tagged reviews first. Thus, I saw fit to discard the negatively tagged reviews for the sake of simplifying my analysis and reducing my dataset to a more manageable size. In the end, this technique resulted in my working dataset being just under 10,000 reviews.

Next, I classified 2,000 of the reviews as either sarcastic or not sarcastic for use with my Naive Bayes Classifier. I then used those 2,000 reviews to train my classifier and then used the classifier to sort the remaining 8,000 reviews into sarcastic and non-sarcastic sets.

Next, I classified an additional 3,000 reviews and ran the Viterbi algorithm over them. I then saved the sentence structure outputs of these reviews by their classification. My plan was to use this to discern some kind of pattern in sentence structure shared by each classification of review.

Finally, I ran a simple test over the set of 3,000 reviews to calculate statistics about the punctuation and capitalization in sarcastic and non-sarcastic reviews to discern if there is some correlation for predicting sarcastic text.

5.2 Results

As I mentioned earlier, I was met with mixed results. It turned out that a simple binary classifier when appropriately trained is reasonably effective when flagging fake and sarcastic reviews. There was a notable gap in the reviews flagged, in that reviews which followed the formula of: "x bad thing happened, 10/10 would x again" were marked as non-sarcastic. This is likely due to real reviews also including a large proportion of the appearances of the "word" 10/10. However, besides this caveat it is reasonable to assume that another experiment with a larger dataset, perhaps one proportional to the original dataset size would yield a much stronger result.

My sentence structure analysis was much less successful. It turns out, that the variety of slang and sentence structure variations (or lack thereof) contributed to a breakdown in my implementation of the Viterbi algorithm. Essentially, the way I handled unknowns was to replace the unknown word with a placeholder marker. Unfortunately due to the sheer magnitude of the unknowns in the dataset for both classifications the sentence structure markers were all some variation of "unknown, part of speech" or "part of speech, unknown." Therefore, I was forced to conclude that it was inconclusive whether or not there is a correlation between sentence structure and sarcasm in text.

My analysis of Punctuation and Capitalization yielded some basic results that were not quite definitive enough to be conclusive. I calculated the average punctuation marks per sentence in a review and the average capital letters per review. I calculated these numbers for each tag. For sarcastic reviews I came up with the following numbers: 1.8728 Punctuation marks per sentence on average and 0.1765 capitalized letters per review on average. For non-sarcastic reviews I came up with: 1.4658 Punctuation marks per sentence and 0.0003 capitalized letters per review. From this we can see that there is clearly more of a correlation between capitalized letters in sarcastic reviews than non-sarcastic reviews. However it is possible this comes from the reasonably small sample size. It is unlikely that 3,000 reviews is not large enough of a sample size, however more data points could change the outcome of the data. Additionally, the same is true for Punctuation. The two punctuation values are significantly closer between the two classifications but there still stands the possibility that there is some correlation there. This test is by no means conclusive in either direction. However, it does open the possibility for further development of a sarcasm predicting model.

6. CONCLUSION

In Conclusion, there is some merit for a Naive Bayes Classifier working with a binary model to interpret text and identify sarcasm despite the difficult nature of identifying sarcasm in text. However, due to sentence structure issues and the way people write and format their reviews it is unreliable to analyze anything with sentence structure analysis unless heavy cleaning is done. For a model such as this to be practical this would have to be an automated process which I was unable to design. There also may be some correlation between punctuation and capitalization and sarcasm in text but the correlation is in no way guaranteed or definitive.

All in all, I learned a lot about the power of a simple classifier. I identified some possible explanatory variables which could assist the classifier in identifying sarcasm in text. I also learned a great deal about the nature of online data. The difficulties of the data set made this project a great deal more difficult than it could have been. However, I believe I did a reasonable job with the difficult dataset I chose.

7.1 Future Plans

As mentioned in prior sections I did not accomplish everything I wanted to in this project. There are many ideas I had that I was not able to fully flesh out to have any results to write about in this report. To make cleaning the data easier I would have liked to create some kind of noise reduction algorithm. However, the issue being the random strings of non standard letters and non-English reviews as opposed to slang and misspelled reviews made this quite the difficult

task. While I opted to do this by hand I would have liked to have been able to create my own algorithm to handle this.

For my general analysis, I would have liked to use a larger sample set (as in greater than the 10,000 I actually looked at) and used a wider range of classifiers instead of the simple binary model I worked with.

Finally, I would have liked to finalize some kind of algorithm which works with the Naive Bayes classifier to take into account capitalization and punctuation by means of a weighting system.

7. ACKNOWLEDGEMENTS

I would like to thank Scott Burger for providing me with the datasets used in his analysis of steam reviews as well as the code he used to generate them.

8. REFERENCES

[1] R. Bais, P. Odek, and S. Ou. Sentiment classification on steam reviews. 2017.

[2] S. V. Burger. Sentiment mining of steam user reviews. 2017.

[3] A. Sobkowicz and W. Stokowiec. Steam review dataset - new, large scale sentiment dataset. 2016.

APPENDIX

A. Implementation of the Naive Bayes Algorithm

```
def train(trainingSet, classes):
    """
    train
        Takes in a set of documents and a list of classes and trains a Naive Bayes classifier.
    Inputs:
        trainingSet: a set of tuples (text,class) to be used for training
        classes: a list of the different classes the classifier should be trained over
    Outputs:
        logPrior: a calculated probability from the number of documents in a class over all documents
        logLikelihood: a dictionary containing calculated probability for every word for each class
    """
    logLikelihood = {} # maps words to the probability they will be in a document of a certain class {class:{word:proba
    classlen = {} # maps a class to the number of documents in that class {class:number}
    logPrior= {}
    vocab = {}

    # set the dictionaries in train to empty dictionaries and count the number of documents in each class
    for i in trainingSet:
        logLikelihood[i[1]] = {}
        if i[1] in classlen:
            classlen[i[1]] += 1
        else:
            classlen[i[1]] = 1

    #calculate the total number of documents
    docNum = 0
    for key in classlen:
        docNum = classlen[key] + docNum

    #calculate logPrior for each class
    for key in classlen:
            logPrior[key] = math.log(classlen[key]/docNum)

    for doc in trainingSet:
        for word in nltk.regexp_tokenize(doc[0].lower(),r'[a-z]+'):
            if(word not in vocab):
                vocab[word] = 0
                for c in classes:
                    logLikelihood[c][word] = 0
            # add 1 to a word's probability for every document it's in, we will divide this by the number of documents
            logLikelihood[doc[1]][word] +=1


    #calculate probabilities
    for i in logLikelihood:
        for w in logLikelihood[i]:
            logLikelihood[i][w] = math.log((logLikelihood[i][w]+1)/(classlen[i]+1))
    vocab = list(vocab.keys())
    return logPrior,logLikelihood,vocab

def test(testDoc, logPrior, logLikelihood, classes, vocab):
    """
    test
        Using a trained classifier tests a given document and returns the perceived class of the document
    Inputs:
        testDoc: document to be tested
        logPrior: calculated logPrior values from train
        logLikelihood: calculated logLikelihood values from train
        classes: list of the classes to be tested
        vocab: list of all words known by the classifier
    Outputs:
        classifier: returns the calculated classifier of the document
```

```
    """
    total = {}
    #break text into words
    words = nltk.regexp_tokenize(testDoc[0].lower(),r'[a-z]+')

    #for every classifier and every word calculate the total likelihood values
    for c in classes:
        total[c] = logPrior[c]
        for w in words:
            if w in vocab:
                total[c] = total[c] + logLikelihood[c][w]

    #whichever classifier has the higher likelihood is the winner
    bigger = classes[0]
    for c in classes:
        if(total[c] > total[bigger]):
            bigger = c
    return bigger

def testCorpus(testSet, logPrior, logLikelihood, classes, vocab):
    """
    testCorpus
        Uses a trained Naive Bayes classifier to test an entire set of documents.
    Inputs:
        testSet: set of documents to be tested
        logPrior: calculated logPrior values from train
        logLikelihood: calculated logLikelihood values from train
        classes: list of the classes to be considered
        vocab: list of all words known by the classifier
    Outputs:
        recall: a calculated value representing the percentage of things that were in the input that were correctly ide
        precision: a calculated value representing the percentage of things detected
    """
    #initialize counters
    falsenegs = []
    falseposs = []
    truepos = 0
    falsepos = 0
    falseneg = 0
    trueneg = 0
    #i = 1;
    for doc in testSet: #run the classifier for each doc
        #print("Document number:", i)
        #i+=1
        result = test(doc,logPrior,logLikelihood,classes,vocab)
        #if the result matches log a true positive or negative
        if(result == doc[1]):
            if(result == classes[1]):
                trueneg +=1
            else:
                truepos +=1
        else: #otherwise record a false positive or negative
            if(result == classes[1]):
                falseneg +=1
                falsenegs.append(doc)
            else:
                falsepos+=1
                falseposs.append(doc)

    #do final calculations
    recall = truepos / (falseneg + truepos)
    precision = truepos / (falsepos + truepos)
```

```
    return precision,recall,falsenegs,falseposs
```

B. Usage of the Bayes Classifier

```
def test1(testDoc, logPrior, logLikelihood, classes, vocab):
    """
    test
        Using a trained classifier tests a given document and returns the perceived class of the document
    Inputs:
        testDoc: document to be tested
        logPrior: calculated logPrior values from train
        logLikelihood: calculated logLikelihood values from train
        classes: list of the classes to be tested
        vocab: list of all words known by the classifier
    Outputs:
        classifier: returns the calculated classifier of the document
    """
    total = {}
    #break text into words
    words = nltk.regexp_tokenize(testDoc.lower(),r'[a-z]+')

    #for every classifier and every word calculate the total likelihood values
    for c in classes:
        total[c] = logPrior[c]
        for w in words:
            if w in vocab:
                total[c] = total[c] + logLikelihood[c][w]

    #whichever classifier has the higher likelihood is the winner
    bigger = classes[0]
    for c in classes:
        if(total[c] > total[bigger]):
            bigger = c
    return bigger

def sortDocs(testSet, logPrior, logLikelihood, classes, vocab):
    """
    sortDocs
        Uses a trained Naive Bayes Classifier to sort documents into what it thinks its class should be
    Inputs:
        testSet: set of documents to be tested
        logPrior: calculated logPrior values from train
        logLikelihood: calculated logLikelihood values from train
        classes: list of the classes to be considered
        vocab: list of all words known by the classifier
    Outputs:
        poslist: list of all documents the classifier marks as positive
        neglist: list of all documents the classifier marks as negative
    """
    poslist = []
    neglist = []
    for doc in testSet: #run the classifier for each doc
        result = test1(doc,logPrior,logLikelihood,classes,vocab)
        if(result == classes[1]):
            neglist.append(doc)
        else:
            poslist.append(doc)
    return(poslist,neglist)
```

C. Implementation of the Viterbi Algorithm

```
#==============================================================================
#    Viterbi
```

```
#            Given a sentance of words and the two training matrices determines
#            the part of speech of each word using the viterbi algorithm
#=============================================================================
def Viterbi(A, B, x):
    '''Viterbi decoding algorithm (logarithmic probabilities)
            A is the transition probability matrix (includes start state 0)
            B is the emission probability matrix (includes 0's for state 0)
            - stored as a dictionary of (state,symbol) pairs
            x is the sequence of observations'''

    n = len(x)                                          # number of observations
    m = len(A)                                          # number of states (including start state)
    v = numpy.zeros((m, n + 1))  - numpy.inf            # matrix of v[q,i] values, initialized to -infinity
    v[0, 0] = 0                                         # log prob of starting in the start state is 0
    s = numpy.zeros((m, n + 1), dtype = numpy.int32)  # s[q,i] = optimal previous state for v[q,i]

    ##############################################################################
    for t in range(1,n+1):
        for j in range(1,m):
            maxVal = -numpy.inf
            maxi = 0
            if (j,x[t-1]) not in B:        #if the given word is not in the vocabulary change it to unknown
                word = "unk"
            else:
                word = x[t-1]
            for i in range(m):
                temp = v[i,t-1] + A[i,j]
                if temp > maxVal:
                    maxVal = temp
                    maxi = i
            v[j,t] = maxVal + B[(j,word)]
            s[j,t] = maxi
    ##############################################################################

    maxS = numpy.argmax(v[:,n])  # index (state) of the maximum value in the last column of v
    maxP = v[maxS, n]            # the maximum value in the last column of v

    # Construct the optimal sequence of states by backtracking.

    path = numpy.zeros(n + 1, dtype = numpy.int32)  # init state sequence to be a list of zeroes
    path[n] = maxS                                  # start in state maxS
    for i in range(n - 1, -1, -1):                  # and follow the "links" backwards
        path[i] = s[path[i + 1], i + 1]

    return maxP, path

#=============================================================================
#    train
#        Creates the matrices containing the training data for use with our
#        vitterbi algorithm.
#=============================================================================
def vtrain(tagged_sents):
    '''
    Inputs:
        tagged_sents: A list of tuples.
            element[0]: The word in the sentance.
            element[1]: The part of speech of the word.
    Outputs:
        Alog: A matrix representing transition probabilities
        Blog: A dictionary representing observation probabilities
        tagsfromindex: A dictionary containing the part of speech for each
        position in the Alog matrix
    '''
```

```
    tags = {} # will be a dictionary of dictionaries counting the times a tag appears given a previous tag
    wordcounts ={} # a dictionary of words with values being the number of times a given word appears in the training s
    btuples = {} # a dictionary of tag,word tuples with values being the number of times the given tag/word combo appea
    tagcounts = {} # a dictionary of tags containing the number of times the given tag appears

    for tagged_sent in tagged_sents:
        prev = '<s>'

        for word in tagged_sent: # iterate over words in the sentance
            if(word[0] not in wordcounts): # count how many times each word appears
                wordcounts[word[0]] = 1
            else:
                wordcounts[word[0]] += 1

            if(word[1] not in tagcounts): # count how many times each tag appears
                tagcounts[word[1]] = 1
            else:
                tagcounts[word[1]] += 1

            if((word[1],word[0]) not in btuples): # count how many times each tag,word combo appears
                btuples[(word[1],word[0])] = 1
            else:
                btuples[(word[1],word[0])] +=1

            if prev not in tags: # count how many times each tag appeares given the previous tag
                tags[prev] = {}
                tags[prev][word[1]] = 1
            else:
                if(word[1] not in tags[prev]):
                    tags[prev][word[1]] =1
                else:
                    tags[prev][word[1]] += 1
            prev = word[1]

    labels = list(tags.keys())                # gives a list of tags in alphabetical order
    n = len(labels)
    A = numpy.zeros((n, n), dtype = numpy.float) # initialize our matrix
    indexes = {} # create a dictionary indexed by indicies in A which contains the tags it coorisponds to
    tagsfromindex ={} # create a dictionary indexed by tags containing their index in A

    row = 0 # start on the first row of the matrix
    for prev in labels: #for each previous tag
        indexes[prev] = row
        tagsfromindex[row] = prev
        col = 0
        for curr in labels: #for each current tag
            if curr not in tags[prev]: #if the combo did not appear give 0 probability
                A[row][col] = 0
            else: # otherwise give count/appearences of tag probability
                A[row][col] = float(tags[prev][curr])/float(tagcounts[curr])
            col += 1
        row += 1
    Alog = numpy.log(A) # take the logarithm of each entry to normalize the matrix

    B = {} # create b dictionary
    for key in labels: #insert unknown key,word pairs
        B[(indexes[key],"unk")] = 0
#=========================================================================
    UNKNOWNCUTOFF = 1 #unknowncutoff is set to 1 currently
#=========================================================================
    vocab = list(wordcounts.keys()) # vocabulary is the list of all unique words in the training set
    for word in vocab: # for every unique word
        for key in labels: # for every unique key
```

```
            if(wordcounts[word] <= UNKNOWNCUTOFF): # AND the word appeared less times than UNKNOWN Cutoff
                if((key,word) in btuples): # if the key/word combo occured
                    B[(indexes[key],"unk")] += float(btuples[(key,word)]) # contribute the word/tag combo's appearences
            else:
                if((key,word) in btuples): # if the key/word combo occured
                    B[(indexes[key],word)] = float(btuples[(key,word)])/float(tagcounts[key])
                    #calculate the probability by tag,word appearences / tag appearences
                else: # if the combo did not appear then there is zero probability
                    B[(indexes[key],word)] = 0
    for key in labels: # for every key
        if(not key == "<s>"): # besides <s>
            B[(indexes[key],"unk")] = float(B[(indexes[key],"unk")])/float(tagcounts[key]) # divide the unknown count b
        else:
            B[(indexes[key],'unk')] = 0 # set the probability to 0 if it is <s>
    Blog = {(s, t): numpy.log(B[(s,t)]) for (s, t) in B} # normalize matrix by taking the log of each element
    return Alog, Blog, tagsfromindex
```

D. Usage of the Viterbi Algorithm

```
#=============================================================================
#    test
#         Given trained matrices and a method to get indices from tags will
#        run the viterbi algorithm over a given testing set and report
#        accuracy
#=============================================================================
def test(Alog, Blog, tagsfromindex, test_set):
    '''
    Inputs:
        Alog: Matrix representing transiton probabilities normalized by log
        Blog: Matrix representing observation probabilities normalized by log
        tagsfromindex: Matrix containing tag values indexed by their index in A
        test_set: A set of tagged sentances for testing the Viterbi Algorithm
    Outputs:
        accuracy: The % accuracy of the algorithm on this training set
        errorlist: a list of tuples containing the word, the given tag, and the correct tag
    '''
    sarcasmlist = {}
    nocasmlist = {}
    for review in test_set: #for every review
        sent_detector = nltk.data.load('tokenizers/punkt/english.pickle')
        sentances = sent_detector.tokenize(review[0])
        for sent in sentances:
            sentance = []
            words = nltk.regexp_tokenize(sent,r'[A-z]+')
            prob, path = Viterbi(Alog, Blog, words) # run viterbi on each sentance
            for i in range(len(words)):
                sentance.append(tagsfromindex[path[i+1]])
            sentance = tuple(sentance)
            if(review[1] == 'Yes'):
                if(sentance in sarcasmlist):
                    sarcasmlist[sentance][0]+=1
                else:
                    sarcasmlist[sentance] = [1]
            else:
                if(sentance in nocasmlist):
                    nocasmlist[sentance][0]+=1
                else:
                    nocasmlist[sentance] = [1]
    return sarcasmlist,nocasmlist
```

E. Capitalization and Punctuation Testing

```
def pcavg(puncset):
        sarcasmcavg = 0
```

```
sarcasmpavg = 0
nocasmcavg = 0
nocasmpavg = 0
sarcasmct = 0
nocasmct = 0

for review in puncset:
    reviewpavg = 0
    reviewcavg = 0
    lc = 0
    sc = 0
    text = review[0]
    sent_detector = nltk.data.load('tokenizers/punkt/english.pickle')
    sentences = sent_detector.tokenize(text)

    matcher = r'[A-Z]'
    matcher2 = r'[A-z]'
    matcher3 = r'[.,\/#!$%\^&\*;:{}=\-_`~()<>]'

    lc = len(re.findall(matcher2,text))

    if(lc==0):
        reviewcavg = 0
    else:
        reviewcavg = len(re.findall(matcher,text))/lc

    words = nltk.regexp_tokenize(text,r'[A-z.,\/#!$%\^&\*;:{}=\-_`~()<>]+')

    if(len(sentences)==0):
        reviewpavg = 0
    else:
        reviewpavg = len(re.findall(matcher3,text))/len(sentences)

    if(review[1] == "Yes"):
        sarcasmct = sarcasmct +1
        sarcasmcavg = sarcasmcavg + reviewcavg
        sarcasmpavg = sarcasmpavg + reviewpavg
    else:
        nocasmct = nocasmct +1
        nocasmcavg = nocasmcavg + reviewcavg
        nocasmpavg = nocasmpavg + reviewpavg
sarcasmcavg = sarcasmcavg/sarcasmct
sarcasmpavg = sarcasmpavg/sarcasmct
nocasmcavg = sarcasmcavg/sarcasmct
nocasmpavg = nocasmpavg/nocasmct

return(sarcasmcavg,sarcasmpavg,nocasmcavg,nocasmpavg)
```