

# SYNOPSIS FOR SENTIMENT ANALYSIS

*\*(the outputs shown below are of classification of movie reviews into positive or negative) (sentdex course on youtube)*

## WORK PLAN:

Finding a good dataset having a large amount of separated out positive and negative reviews on movies. The dataset should consist of a folder having sentences of negative anotation and another folder having sentences of positive anotation.

Grouping a sentence with the annotation it conveys and appending it to list known as "documents". All the adjectives used in the dataset are appended to a separate list called "all\_words".

In "all\_words" filtering out the 5000 most used adjectives and storing it in another list known as "word\_features".

Defining a function known as "find\_features" which accepts sentence, separates it into words and checks whether the words are present in "word\_features". Depending on whether the word is present or not the word is grouped with either true or false and is then appended to a list and is returned.

This is done to make the algorithm more efficient. So the algorithm just checks the anotation of the most commonly used adjectives instead of all the adjectives.

Iterating through documents and passing a sentence in find\_features and saving this result in 'featuresets'.

Shuffling the 'featuresets' and then deviding it into two "training\_set" and "testing\_set". "training\_set" shall consist of sufficiently larger number of sentences as compared to "testing\_set". 'featuresets' is shuffled to ensure that the algorithms are trained and tested on both sentences having a positive and a negative anotation.

Training 5 algorithms on the 'training\_set' and checking thier accuracy on the testing set. The algorithms used are Naive Bayes, Bernoulli Naive Bayes and Logistic Regression . After training these algorithms their accuracy is tested using the 'testing\_set'.

Creating a class known as 'VoteClassifier' which takes in a bunch of algorithms and returns the majority result of the algorithms. It also calculates the confidence. So to explain confidence, it is like taking advice from a group of people and going with the advice of

the majority. That is why an odd number of algorithms needs to be used.

Once the class 'VoteClassifier' has been defined, pass the algorithms to it. 'VoteClassifier' shall have a function known as 'classify' which shall take in a sentence and test it on the algorithms and return the result which is agreed on by majority of the algorithms.

Running the program once and pickling all the algorithms so that the time required for running decreases drastically.

Defining a function known as 'sentiment' which shall take in any sentence and pass it into the 'classify' function of 'VotedClassifier'. 'sentiment' returns the anotation of that sentence according to majority vote of the algorithms and also returns the confidence.

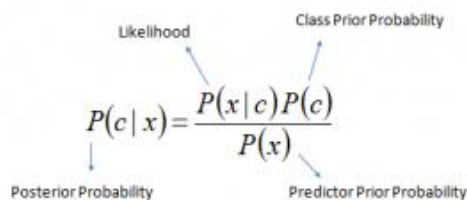
Now using twitter api and passing tweets into the 'sentiment' function and and storing the anotation of each tweet in a .txt file and then plotting out the data.

## ALGORITHMS USED:

### Naive Bayes Algorithm:

It is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'.



The diagram shows the formula for Bayes' Theorem: 
$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$
 with arrows pointing from labels to parts of the formula: 'Likelihood' points to  $P(x|c)$ , 'Class Prior Probability' points to  $P(c)$ , 'Posterior Probability' points to  $P(c|x)$ , and 'Predictor Prior Probability' points to  $P(x)$ .

$$P(c|X) = P(x_1|c) \times P(x_2|c) \times \dots \times P(x_n|c) \times P(c)$$

### Bernoulli Naive Bayes Algorithm:

BernoulliNB implements the naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions; i.e., there may be multiple features but each one is assumed to be a binary-valued (Bernoulli, boolean) variable.

The decision rule for Bernoulli naive Bayes is based on:

$$P(x_i | y) = P(i | y)x_i + (1 - P(i | y))(1 - x_i)$$

## **Logistic Regression:**

Image result for logistic regression algorithm

Logistic Regression is a Machine Learning algorithm which is used for the classification problems, it is a predictive analysis algorithm and based on the concept of probability.

## **PROBLEMS FACED:**

Initially planned to test adverbs, verbs and adjectives. But on passing a few sentences found a mistake. So my algorithm was returning 'neg' for "this was awesome" and was returning 'pos' for 'this awesome'. Since 'was' was used a lot more in negative sentences as compared to 'awesome' used in positive sentences, it was returning 'neg' for "this was awesome".

Solved the problem by just filtering the adjectives.

Tried using lemmatizing. Initially lemmatizing increased the accuracy but it had slowed the model down due to which the algorithm could not keep up with the tweets returned from twitter API and an error used to occur.

Hence did not use lemmatizing.

Initially was using corpora for data sets to train and test the algorithms, but required a larger dataset.

## **EXPLANATION:**

```

short_pos = open("short_reviews/positive.txt","r").read()
short_neg = open("short_reviews/negative.txt","r").read()

# move this up here
all_words = []
documents = []

# j is adjct, r is adverb, and v is verb
#allowed_word_types = ["J", "R", "V"]
allowed_word_types = ["J"]

for p in short_pos.split('\n'):
    documents.append( (p, "pos") )
    words = word_tokenize(p)
    pos = nltk.pos_tag(words)
    for w in pos:
        if w[1][0] in allowed_word_types:
            all_words.append(w[0].lower())

for p in short_neg.split('\n'):
    documents.append( (p, "neg") )
    words = word_tokenize(p)
    pos = nltk.pos_tag(words)
    for w in pos:
        if w[1][0] in allowed_word_types:
            all_words.append(w[0].lower())

```

"allowed\_word\_types" is used to filter out the adjectives from all the sample text present.

short\_pos.split('\n') returns a list of sentences. Since p is iterating through 'short\_pos.split('\n')' it is gonna be a sentence. The sentence is then made into a list of words using word\_tokenize. Then '.pos\_tag(words)' returns a list of tuples, where each tuple contains the word and the role played by the word in a sentence. Then this is passed through another for loop which just filters out the adjectives.

```

all_words = nltk.FreqDist(all_words)

word_features = list(all_words.keys())[:5000]

def find_features(document):
    words = word_tokenize(document)
    features = {}
    for w in word_features:
        features[w] = (w in words)

    return features

featuresets = [(find_features(review), category) for (review, category) in documents]

random.shuffle(featuresets)
print(len(featuresets))

testing_set = featuresets[10000:]
training_set = featuresets[:10000]

classifier = nltk.NaiveBayesClassifier.train(training_set)
print("Original Naive Bayes Algo accuracy percent:", (nltk.classify.accuracy(classifier, testing_set))*100)
classifier.show_most_informative_features(15)

```

The words are then arranged in order of the number of times they are repeated using 'nltk.FreqDist(all\_words)'. The 5000 most used adjectives are then stored it in 'word\_features' .

Function 'find\_features' accepts a sentence converts it into a list words using 'word\_tokenize'. Then checking whether the words are a part of 'word\_features' and accordingly assinging the word true or false and returning.

Shuffling featuresets to insure that the training set and testing set have both positive and negative sentences.

Then training all the algorithms followed by pickling to make it faster.

```

class VoteClassifier(ClassifierI):
    def __init__(self, *classifiers):
        self._classifiers = classifiers

    def classify(self, features):
        votes = []
        for c in self._classifiers:
            v = c.classify(features)
            votes.append(v)
        return mode(votes)

    def confidence(self, features):
        votes = []
        for c in self._classifiers:
            v = c.classify(features)
            votes.append(v)

        choice_votes = votes.count(mode(votes))
        conf = choice_votes / len(votes)
        return conf

```

Creating a class known as 'VoteClassifier' which takes in a bunch of algorithms and returns the majority result of the algorithms. It also calculates the confidence. So to explain confidence, it is like taking advice from a group of people and going with the advice of the majority. That is why an odd number of algorithms needs to be used.

```

def sentiment(text):
    feats = find_features(text)
    return voted_classifier.classify(feats), voted_classifier.confidence(feats)

```

Making it callable via a function and then just passing in tweets from the twitter api.