Report

(Sentiment Analysis)

Hello everyone, we wrote a report about our project.

The project is about classifying tweets wether they’re positive or negative by using sentiment analysis with nltk and other packages, we used also twitter app to stream live tweets by give them a label (pos or neg) and the score of confidence (accuracy).

1.classification of text :

First of all, we started with text classification after importing all the packages that we need.

To do this, we started by using a dataset of some positive and negative tweets. From there we tried to use words as "features" which are a part of either a positive or negative tweet. This means we can train and test with this data.

Next, we used random to shuffle our data. This is because we're going to be training and testing. If we left them in order, chances are we'd train on all of the negatives, some positives, and then test only against positives. We don't want that, so we shuffled the data.

Combining classifier algorithms is a common technique, done by creating a sort of voting system, where each algorithm gets one vote, and the classification that has the most votes is the chosen one. To do this, we want our new classifier to act like a typical NLTK classifier, with all of the methods. By using object-oriented programming, we can just be sure to inherit from the NLTK classifier class.

We're calling our class the VoteClassifier, and we're inheriting from NLTK's ClassifierI. Next, we're assigning the list of classifiers that are passed to our class to self._classifiers.

Next, we want to go ahead and create our own classify method. We want to call it classify, so that we can invoke classify later on, like a traditional NLTK classifier would allow.

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

all we're doing here is iterating through our list of classifier objects. Then, for each one, we ask it to classify based on the features. The classification is being treated as a vote. After we are done iterating, we then return the mode(votes), which is just returning the most popular vote.

This is all we really need, but I think it would be useful to have another parameter, confidence. Since we have algorithms voting, we can also tally the votes for and against the winning vote, and call this "confidence." For example, 3/5 votes for positive is weaker than 5/5 votes. As such, we can literally return the ratio of votes as a sort of confidence indicator. Here's our confidence method:

```
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
```

Now let's put everything together, this is the class:

```
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
```

2.word as features :

we're going to be build off a feature list of words from positive tweets and words from the negative ones to hopefully see trends in specific types of words in positive or negative tweets.

```
word_features = list(all_words.keys())[:3000]
```

Mostly the same as before, only with now a new variable, word_features, which contains the top 3,000 most common words. Next, we're going to build a quick function that will find these top 3,000 words in our positive and negative tweets, marking their presence as either positive or negative:

```
def find_features(document):

    words = word_tokenize(document)

    features = {}

    for w in word_features:

        features[w] = (w in words)


    return features
```

Then we can do this for all of our documents, saving the feature existence booleans and their respective positive or negative categories by doing:

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

now that we have our features and labels, what is next? Typically, the next step is to go ahead and train an algorithm, then test it. So, let's start with the Naive Bayes classifier.

3.Naive Bayes classifier

The algorithm that we're going to use first is the Naive Bayes classifier. This is a pretty popular algorithm used in text classification, so it is only fitting that we

try it out first. Before we can train and test our algorithm, however, we need to go ahead and split up the data into a training set and a testing set.

We could train and test on the same dataset, but this would present us with some serious bias issues, so we should never train and test against the exact same data. To do this, since we've shuffled our data set, we'll assign the first 10000 shuffled tweets, consisting of both positive and negative tweets, as the training set. Then, we can test against the last 100 to see how accurate we are.

This is called supervised machine learning, because we're showing the machine data, and telling it "hey, this data is positive," or "this data is negative." Then, after that training is done, we show the machine some new data and ask the computer, based on what we taught the computer before, what the computer thinks the category of the new data is.

We can split the data with:

```
# set that we'll train our classifier with
training_set = featuresets[:10000]

# set that we'll test against.
testing_set = featuresets[10000:]
```

Next, we can define, and train our classifier like:

```
classifier = nltk.NaiveBayesClassifier.train(training_set)
```

First we just simply are invoking the Naive Bayes classifier, then we go ahead and use .train() to train it all in one line.

the reason why we can "test" the data is because we still have the correct answers. So, in testing, we show the computer the data without giving it the correct answer. If it guesses correctly what we know the answer to be, then the computer got it right. Given the shuffling that we've done, we might come up with varying accuracy, but we should see something from 60-75% on average.

Next, we can take it a step further to see what the most valuable words are when it comes to positive or negative reviews:

```
classifier.show_most_informative_features(15)
```

This is the output:

```
Most Informative Features
```

```
              engrossing = True              pos : neg    =     20.4 :
1.0
                 generic = True              neg : pos    =     16.9 :
1.0
                mediocre = True              neg : pos    =     16.2 :
1.0
                 routine = True              neg : pos    =     15.6 :
1.0
                  boring = True              neg : pos    =     13.5 :
1.0
                    flat = True              neg : pos    =     13.3 :
1.0
              inventive = True               pos : neg    =     13.1 :
1.0
                   warm = True               pos : neg    =     12.3 :
1.0
              wonderful = True               pos : neg    =     11.9 :
1.0
              realistic = True               pos : neg    =     11.1 :
1.0
               mindless = True               neg : pos    =     10.9 :
1.0
                  stale = True               neg : pos    =     10.9 :
1.0
                 stupid = True               neg : pos    =     10.5 :
1.0
                   dull = True               neg : pos    =     10.3 :
1.0
               powerful = True               pos : neg    =     10.2 :
1.0
```

What this tells us is the ratio of occurrences in negative to positive, or visa versa, for every word. So here, we can see that the term " engrossing" appears 20.4 more times as often in positive tweets as it does in negative tweets. Now, let's say we were totally content with our results, and we wanted to move forward, maybe using this classifier to predict things right now. It would be very impractical to train the classifier, and retrain it every time we needed to use it. As such, we can save the classifier using the pickle module.

Training classifiers and machine learning algorithms can take a very long time, especially if we're training against a larger data set. Ours is actually pretty small, instead, what we can do is use the Pickle module to go ahead and serialize our classifier object, so that all we need to do is load that file in real quick. This is how we do it:

```
save_classifier = open("naivebayes.pickle","wb")
pickle.dump(classifier, save_classifier)
save_classifier.close()
```

This opens up a pickle file, preparing to write in bytes some data. Then, we use pickle.dump() to dump the data. The first parameter to pickle.dump() is what

are you dumping, the second parameter is where are you dumping it. After that, we close the file as we're supposed to, and that is that, we now have a pickled, or serialized, object saved in our script's directory!

All we need to do now is read it into memory, which will be about as quick as reading any other ordinary file. To do this:

```
classifier_f = open("naivebayes.pickle", "rb")
classifier = pickle.load(classifier_f)
classifier_f.close()
```

Here, we do a very similar process. We open the file to read as bytes. Then, we use pickle.load() to load the file, and we save the data to the classifier variable. Then we close the file, and that is that. We now have the same classifier object as before!

Now, we can use this object, and we no longer need to train our classifier every time we wanted to use it to classify.

4.Scikit-Learn classifiers :

We're probably not too content with the 60-75% accuracy we're getting. That's why we're gonna use other classifiers from sklearn module to use it we need to import it first:

```
from nltk.classify.scikitlearn import SklearnClassifier
```

Now we can use the other classifiers like:

```
MNB_classifier = SklearnClassifier(MultinomialNB())
MNB_classifier.train(training_set)
print("MultinomialNB accuracy
percent:",nltk.classify.accuracy(MNB_classifier, testing_set))
LogisticRegression_classifier =
SklearnClassifier(LogisticRegression())
LogisticRegression_classifier.train(training_set)
print("LogisticRegression_classifier accuracy percent:",
(nltk.classify.accuracy(LogisticRegression_classifier,
testing_set))*100)
```

The next thing we can try is to use all of these algorithms at once. An algo of algos! To do this, we can create another classifier, and make the result of that classifier based on what the other algorithms said. Sort of like a voting system, so we'll just need an odd number of algorithms.

5.choosing classifier with a vote :

```
voted_classifier = VoteClassifier(
```

```
                                    NuSVC_classifier,
                                    LinearSVC_classifier,
                                    MNB_classifier,
                                    BernoulliNB_classifier,
                                    LogisticRegression_classifier)

print("voted_classifier accuracy percent:",
(nltk.classify.accuracy(voted_classifier, testing_set))*100)
def sentiment(text):
    feats = find_features(text)
    return
voted_classifier.classify(feats),voted_classifier.confidence(feat
s)
```

This function, which we're calling "sentiment," takes one parameter, which is text. From there, we break down the features with the find_features function we created long ago. From there, now all we need to do is use our voted_classifier to return not only the classification, but also the confidence in that classification.

Now that we have a sentiment analysis module, we can apply it to just about any text, but preferably short bits of text, like from Twitter! To do this, we're going to combine this tutorial with the Twitter streaming API.

```
from tweepy import Stream
from tweepy import OAuthHandler
from tweepy.streaming import StreamListener
import json
import sentiment_mod as s

#consumer key, consumer secret, access token, access secret.
ckey="your personal info"
csecret="your personal info"
atoken="your personal info"
asecret="your personal info"
class listener(StreamListener):

    def on_data(self, data):

                all_data = json.loads(data)

                tweet = all_data["text"]
                sentiment_value, confidence = s.sentiment(tweet)
                print(tweet, sentiment_value, confidence)

                if confidence*100 >= 80:
                       output = open("twitter-out.txt","a")
                       output.write(sentiment_value)
                       output.write('\n')
                       output.close()
```

```
            return True

    def on_error(self, status):
        print(status)

auth = OAuthHandler(ckey, csecret)
auth.set_access_token(atoken, asecret)

twitterStream = Stream(auth, listener())
twitterStream.filter(track=["happy"])
```

Along with that, we're also saving the results to an output file, twitter-out.txt.