3-Way Sentiment Analysis for Tweets

Overview

In this project, we'll build a 3-way polarity (positive, negative, neutral) classification system for tweets, without using NLTK's in-built sentiment analysis engine.

We'll use a logistic regression classifier, bag-of-words features, and polarity lexicons (both in-built and external). We'll also create our own pre-processing module to handle raw tweets.

Data Used

- training.json: This file contains ~15k raw tweets, along with their polarity labels (1 = positive, 0 = neutral, -1 = negative). We'll use this file to train our classifiers.
- develop.json: In the same format as training.json, the file contains a smaller set of tweets. We'll use it to test the predictions of our classifiers which were trained on the training set.

Preprocessing

The first thing that we'll do is preprocess the tweets so that they're easier to deal with, and ready for feature extraction, and training by the classifiers.

To start with we're going to extract the tweets from the json file, read each line and store the tweets, labels in separate lists.

Then for the preprocessing, we'll:

- **segment** tweets into sentences using an NTLK segmenter
- **tokenize** the sentences using an NLTK tokenizer
- **lowercase** all the words
- **remove twitter usernames** beginning with @ using regex
- **remove URLs** starting with http using regex
- **process hashtags** ,for this we'll tokenize hashtags,and try to break down multi-word hashtags using a MaxMatch algorithm, and the English word dictionary supplied with NLTK.

Let's build some functions to accomplish all this.

In [1]:
```
import json
import re
import nltk

lemmatizer = nltk.stem.wordnet.WordNetLemmatizer()
dictionary = set(nltk.corpus.words.words()) #To be used for MaxMatch

#Function to lemmatize word | Used during maxmatch
def lemmatize(word):
    lemma = lemmatizer.lemmatize(word,'v')
    if lemma == word:
        lemma = lemmatizer.lemmatize(word,'n')
    return lemma

#Function to implement the maxmatch algorithm for multi-word hashtags
def maxmatch(word,dictionary):
    if not word:
        return []
    for i in range(len(word),1,-1):
        first = word[0:i]
        rem = word[i:]
        if lemmatize(first).lower() in dictionary: #Important to lowercase lemmatized words before comparing in dictionary.
            return [first] + maxmatch(rem,dictionary)
    first = word[0:1]
    rem = word[1:]
    return [first] + maxmatch(rem,dictionary)

#Function to preprocess a single tweet
def preprocess(tweet):

    tweet = re.sub("@\w+","",tweet).strip()
    tweet = re.sub("http\S+","",tweet).strip()
    hashtags = re.findall("#\w+",tweet)

    tweet = tweet.lower()
    tweet = re.sub("#\w+","",tweet).strip()

    hashtag_tokens = [] #Separate list for hashtags

    for hashtag in hashtags:
        hashtag_tokens.append(maxmatch(hashtag[1:],dictionary))

    segmenter = nltk.data.load('tokenizers/punkt/english.pickle')
    segmented_sentences = segmenter.tokenize(tweet)

    #General tokenization
    processed_tweet = []

    word_tokenizer = nltk.tokenize.regexp.WordPunctTokenizer()
    for sentence in segmented_sentences:
        tokenized_sentence = word_tokenizer.tokenize(sentence.strip())
        processed_tweet.append(tokenized_sentence)

    #Processing the hashtags only when they exist in a tweet
    if hashtag_tokens:
        for tag_token in hashtag_tokens:
            processed_tweet.append(tag_token)

    return processed_tweet

#Custom function that takes in a file, and passes each tweet to the preprocessor
def preprocess_file(filename):
    tweets = []
    labels = []
    f = open(filename)
    for line in f:
        tweet_dict = json.loads(line)
        tweets.append(preprocess(tweet_dict["text"]))
        labels.append(int(tweet_dict["label"]))
    return tweets,labels
```

Before we run preprocess our training data, let's see how well the maxmatch algorithm works.

In [2]:
```
maxmatch('wecan',dictionary)
```

Out[2]:
```
['we', 'can']
```

Let's try feeding it something harder than that.

In [3]:
```
maxmatch('casestudy',dictionary)
```

Out[3]:
```
['cases', 'tu', 'd', 'y']
```

As we can see from the above example, it incorrectly breks down the word 'casestudy', by returning 'cases', instead of 'case' for the first iteration., which would have been a better output. This is because it *greedily* extract 'cases' first.

For an improvement, we can implement an algorithm that better counts the total number of successful matches in the result of the maxmatch process, and return the one with the highest successful match count.

Let's run our preprocessing module on the raw training data.

In [9]:
```
#Running the basic preprocessing module and capturing the data (maybe shift to the next block)
train_data = preprocess_file('data/sentiment/training.json')
train_tweets = train_data[0]
train_labels = train_data[1]
```

Let's print out the first couple processed tweets:

In [10]:
```
print train_tweets[:2]
```

```
[[[u'dear', u'the', u'newooffice', u'for', u'mac', u'is', u'great', u'and', u'all', u',', u'but', u'no', u'lync', u'update', u'?'], [u'c', u"'", u'mon', u'.']], [[u'how', u'about', u'you', u'make', u'a', u'system', u'that', u'doesn', u"'", u't', u'eat', u'my', u'friggin', u'discs', u'.'], [u'this', u'is', u'the', u'2nd', u'time', u'this', u'has', u'happened', u'and', u'i', u'am', u'so', u'sick', u'of', u'it', u'!']]]
```

Hmm.. we can do better than that to make sense of what's happening. Let's write a simple script to that'll run the preprocessing module on a few tweets, and print the original and processed results, side by side; if it detects a multi-word hashtag.

In [12]:
```
#Printing examples of multi-word hashtags (Doesn't work for multi sentence tweets)
f = open('data/sentiment/training.json')
count = 1
for index,line in enumerate(f):
    if count >5:
        break
    original_tweet = json.loads(line)["text"]
    hashtags = re.findall("#\w+",original_tweet)
    if hashtags:
        for hashtag in hashtags:
            if len(maxmatch(hashtag[1:],dictionary)) > 1:
                #If the length of the array returned by the maxmatch function is greater than 1,
                #it means that the algorithm has detected a hashtag with more than 1 word inside.
                print str(count) + ". Original Tweet: " + original_tweet + "\nProcessed tweet: " + str(train_tweets[index]) + "\n"
                count += 1
                break
```

```
1. Original Tweet: If I make a game as a #windows10 Universal App. Will #xboxone owners be able to download and play it in November? @majornelson @Microsoft
Processed tweet: [[u'if', u'i', u'make', u'a', u'game', u'as', u'a', u'universal', u'app', u'.'], [u'will', u'owners', u'be', u'able', u'to', u'download', u'and', u'play', u'it', u'in', u'november', u'?'], [u'windows', u'1', u'0'], [u'x', u'box', u'one']]

2. Original Tweet: Microsoft, I may not prefer your gaming branch of business. But, you do make a damn fine operating system. #Windows10 @Microsoft
Processed tweet: [[u'microsoft', u',', u'i', u'may', u'not', u'prefer', u'your', u'gaming', u'branch', u'of', u'business', u'.'], [u'but', u',', u'you', u'do', u'make', u'a', u'damn', u'fine', u'operating', u'system', u'.'], [u'Window', u's', u'1', u'0']]

3. Original Tweet: @MikeWolf1980 @Microsoft I will be downgrading and let #Windows10 be out for almost the 1st yr b4 trying it again. #Windows10fail
Processed tweet: [[u'i', u'will', u'be', u'downgrading', u'and', u'let', u'be', u'out', u'for', u'almost', u'the', u'1st', u'yr', u'b4', u'trying', u'it', u'again', u'.'], [u'Window', u's', u'1', u'0'], [u'Window', u's', u'1', u'0', u'fail']]

4. Original Tweet: @Microsoft 2nd computer with same error!!! #Windows10fail Guess we will shelve this until SP1! http://t.co/QCcHlKuy8Q
Processed tweet: [[u'2nd', u'computer', u'with', u'same', u'error', u'!!!'], [u'guess', u'we', u'will', u'shelve', u'this', u'until', u'sp1', u'!'], [u'Window', u's', u'1', u'0', u'fail']]

5. Original Tweet: Sunday morning, quiet day so time to welcome in #Windows10 @Microsoft @Windows http://t.co/7VtvAzhWmV
Processed tweet: [[u'sunday', u'morning', u',', u'quiet', u'day', u'so', u'time', u'to', u'welcome', u'in'], [u'Window', u's', u'1', u'0']]
```

That's better! Our pre-processing module is working as intended.

The next step is to convert each processed tweet into a bag-of-words feature dictionary. We'll allow for options to remove stopwords during the process, and also to remove *rare* words, i.e. words occuring less than n times across the whole training set.

In [14]:
```
from nltk.corpus import stopwords

stopwords = set(stopwords.words('english'))

#To identify words appearing less than n times, we're creating a dictionary for the whole training set

total_train_bow = {}

for tweet in train_tweets:
    for segment in tweet:
        for token in segment:
            total_train_bow[token] = total_train_bow.get(token,0) + 1

#Function to convert pre_processed tweets to bag of words feature dictionaries
#Allows for options to remove stopwords, and also to remove words occuring less than n times in the whole training set.
def convert_to_feature_dicts(tweets,remove_stop_words,n):
    feature_dicts = []
    for tweet in tweets:
        # build feature dictionary for tweet
        feature_dict = {}
        if remove_stop_words:
            for segment in tweet:
                for token in segment:
                    if token not in stopwords and (n<=0 or total_train_bow[token]>=n):
                        feature_dict[token] = feature_dict.get(token,0) + 1
        else:
            for segment in tweet:
                for token in segment:
                    if n<=0 or total_train_bow[token]>=n:
                        feature_dict[token] = feature_dict.get(token,0) + 1
        feature_dicts.append(feature_dict)
    return feature_dicts
```

Now that we have our function to convert raw tweets to feature dictionaries, let's run it on our training and development data. We'll also convert the feature dictionary to a sparse representation, so that it can be used by scikit's ML algorithms.

In [22]:
```
from sklearn.feature_extraction import DictVectorizer
vectorizer = DictVectorizer()

#Conversion to feature dictionaries
train_set = convert_to_feature_dicts(train_tweets,True,2)

dev_data = preprocess_file('data/sentiment/develop.json')

dev_set = convert_to_feature_dicts(dev_data[0],False,0)

#Conversion to sparse representations
training_data = vectorizer.fit_transform(train_set)

development_data = vectorizer.transform(dev_set)
```

Classifying

Now, we'll run our data through a decision tree classifier, and try to tune the parameters by using Grid Search over parameter combinations.

In [23]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn import cross_validation
from sklearn.metrics import accuracy_score, classification_report
from sklearn.grid_search import GridSearchCV

#Grid used to test the combinations of parameters
tree_param_grid = [
    {'criterion':['gini','entropy'], 'min_samples_leaf': [75,100,125,150,175], 'max_features':['sqrt','log2',None],
    }
]

tree_clf = GridSearchCV(DecisionTreeClassifier(),tree_param_grid,cv=10,scoring='accuracy')

tree_clf.fit(training_data,train_data[1])

print "Optimal parameters for DT: " + str(tree_clf.best_params_) #To print out the best discovered combination of the parameters

tree_predictions = tree_clf.predict(development_data)

print "\nDecision Tree Accuracy: " + str(accuracy_score(dev_data[1],tree_predictions))
```

```
Optimal parameters for DT: {'max_features': None, 'criterion': 'entropy', 'min_samples_leaf': 75}

Decision Tree Accuracy: 0.487151448879
```

The decision tree classifier doesn't seem to work very well, but we still don't have a benchmark to compare it with.

Let's run our data through a dummy classifier which'll pick the most frequently occuring class as the output, each time.

In [24]:
```
from sklearn.dummy import DummyClassifier

#The dummy classifier below always predicts the most frequent class, as specified in the strategy.
dummy_clf = DummyClassifier(strategy='most_frequent')
dummy_clf.fit(development_data,dev_data[1])
dummy_predictions = dummy_clf.predict(development_data)

print "\nMost common class baseline accuracy: " + str(accuracy_score(dev_data[1],dummy_predictions))
```

```
Most common class baseline accuracy: 0.420448332422
```

We can see that out DT classifier at least performs better than the dummy classifier.

We'll do the same process for logisitc regression classifier now.

In [21]:
```
from sklearn.linear_model import LogisticRegression

log_param_grid = [
    {'C':[0.012,0.0125,0.130,0.135,0.14],
     'solver':['lbfgs'],'multi_class':['multinomial']
    }
]

log_clf = GridSearchCV(LogisticRegression(),log_param_grid,cv=10,scoring='accuracy')

log_clf.fit(training_data,train_data[1])

log_predictions = log_clf.predict(development_data)

print "Optimal parameters for LR: " + str(log_clf.best_params_)

print "Logistic Regression Accuracy: " + str(accuracy_score(dev_data[1],log_predictions))
```

```
Optimal parameters for LR: {'multi_class': 'multinomial', 'C': 0.012, 'solver': 'lbfgs'}
Logistic Regression Accuracy: 0.493165664297
```

To recap what just happened, we created a logistic regression classifier by doing a grid search for the best parameters for C (regularization parameter), solver type, and multi_class handling, just like we did for the decision tree classifier.

We also created a dummy classifier that just picks the most common class in the development set for each prediction.

The table below describes the different classifiers and their accuracy scores.

Classifier	Approx. Accuracy score (in %)
Dummy classifier (most common class)	42
Decision Tree classifier	48.7
Logistic Regression classifier	49.3

As we can see, both classifiers are better than the 'dummy' classifier which just picks the most common class all the time.

Polarity Lexicons

Now, we'll try to integrate external information into the training set, in the form polarity scores for the tweets.

We'll build two automatic lexicons, compare it with NLTK's manually annotated set, and then add that information to our training data.

The **first lexicon** will be built through SentiWordNet. This has pre-calculated scores positive, negative and neutral sentiments for some words in WordNet. As this information is arranged in the form of synsets, we'll just take the most common polarity across its senses (and take neutral in case of a tie).

In [25]:
```
from nltk.corpus import sentiwordnet as swn
from nltk.corpus import wordnet as wn
import random

swn_positive = []

swn_negative = []

#Function supplied with the assignment, not described below.
def get_polarity_type(synset_name):
    swn_synset =  swn.senti_synset(synset_name)
    if not swn_synset:
        return None
    elif swn_synset.pos_score() > swn_synset.neg_score() and swn_synset.pos_score() > swn_synset.obj_score():
        return 1
    elif swn_synset.neg_score() > swn_synset.pos_score() and swn_synset.neg_score() > swn_synset.obj_score():
        return -1
    else:
        return 0


for synset in wn.all_synsets():

    # count synset polarity for each lemma
    pos_count = 0
    neg_count = 0
    neutral_count = 0

    for lemma in synset.lemma_names():
        for syns in wn.synsets(lemma):
            if get_polarity_type(syns.name())==1:
                pos_count+=1
            elif get_polarity_type(syns.name())==-1:
                neg_count+=1
            else:
                neutral_count+=1

    if pos_count > neg_count and pos_count >= neutral_count: #>=neutral as words that are more positive than negative,
                                                             #despite being equally neutral might belong to positive list (explain)
        swn_positive.append(synset.lemma_names()[0])
    elif neg_count > pos_count and neg_count >= neutral_count:
        swn_negative.append(synset.lemma_names()[0])

swn_positive = list(set(swn_positive))
swn_negative = list(set(swn_negative))


print 'Positive words: ' + str(random.sample(swn_positive,5))

print 'Negative Words: ' + str(random.sample(swn_negative,5))
```

```
Positive words: [u'mercy', u'prudent', u'blue_ribbon', u'synergistically', u'controversial']
Negative Words: [u'gynobase', u'anger', u'unservile', u'intestate', u'paresthesia']
```

I'll try and explain what happened.

To calculate the polarity of a synset across its senses, the lemma names were extracted from the synset to get its 'senses'. Then, each of those lemma names were converted to a synset object, which was then passed to the pre-supplied 'get_polarity_type' function. Based on the score returned, the head lemma of the synset object was appended to the relevant list. The head lemma was chosen from the lemma_names, as it best represents the synset object.

As the code above returns a random sample of positive and negative words each time, the words returned when I ran the code the first time (different from the above) were:

Positive words: [u'counterblast', u'unperceptiveness', u'eater', u'white_magic', u'cuckoo-bumblebee'] Negative Words: [u'sun_spurge', u'pinkness', u'hardness', u'unready', u'occlusive']

At first glance, they seem like a better than average sample of negative words, and a worse than average sample of positive ones.

This might be due to the fact that, when looking at a word like 'unperceptiveness', which is a positive word prefixed to convert into a negative one, or an antonym. It's lemmas/senses might contain more positive senses of 'perceptiveness' than negative ones, and has hence been classified as a positive word, which might be wrong.

For the **second lexicon**, we will use the word2vec (CBOW) vectors included in NLTK.

Using a small set of positive and negative seed terms, we will calculate the cosine similarity between vectors of seed terms and another word. We can use Gensim to iterate over words in model.vocab for comparison over seed terms.

After calculating the cosine similarity of a word with both the positive and negative terms, we'll calculate their average, after flipping the sign for negative seeds. A threshold of ±0.03 will be used to determine if words are positive or negative.

In [26]:
```
import gensim
from nltk.data import find
import random

positive_seeds = ["good","nice","excellent","positive","fortunate","correct","superior","great"]
negative_seeds = ["bad","nasty","poor","negative","unfortunate","wrong","inferior","awful"]

word2vec_sample = str(find('models/word2vec_sample/pruned.word2vec.txt'))
model = gensim.models.Word2Vec.load_word2vec_format(word2vec_sample,binary=False)

wv_positive = []
wv_negative = []

for word in model.vocab:
    try:
        word=word.lower()

        pos_score = 0.0
        neg_score = 0.0

        for seed in positive_seeds:
            pos_score = pos_score + model.similarity(word,seed)

        for seed in negative_seeds:
            neg_score = neg_score + model.similarity(word,seed)

        avg = (pos_score - neg_score)/16 #Total number of seeds is 16

        if avg>0.03:
            wv_positive.append(word)
        elif avg<-0.03:
            wv_negative.append(word)
    except:
        pass


print 'Positive words: ' + str(random.sample(wv_positive,5))

print 'Negative Words: ' + str(random.sample(wv_negative,5))
```

```
Positive words: [u'hoping', u'treble', u'revolutionary', u'sumptuous', u'productive']
Negative Words: [u'lawless', u'trudged', u'perpetuation', u'mystified', u'tendency']
```

Again, the code randomises the printed positive and negative words. Tn my first instance, they were:

Positive words: [u'elegant', u'demonstrated', u'retained', u'titles', u'strengthen'] Negative Words: [u'scathingly', u'anorexia', u'rioted', u'blunders', u'alters']

This looks like a great set of both positive negative words, looking at the samples. But let's see how it compares with NLTK's manually annotated set.

The Hu and Liu lexicon included with NLTK, has a list of positive and negative words.

First, we'll investigate what percentage of the words in the manual lexicon are in each of the automatic lexicons, and then, only for those words which overlap and which are not in the seed set, evaluate the accuracy of with each of the automatic lexicons.

In [27]:
```
from nltk.corpus import opinion_lexicon
import math
from __future__ import division

positive_words = opinion_lexicon.positive()
negative_words = opinion_lexicon.negative()

#Calculate the percentage of words in the manually annotated lexicon set, that also appear in an automatic lexicon.
def get_perc_manual(manual_pos,manual_neg,auto_pos,auto_neg):
    return len(set(manual_pos+manual_neg).intersection(set(auto_pos+auto_neg)))/len(manual_pos+manual_neg)*100

print "% of words in manual lexicons, also present in the automatic lexicon"
print "First automatic lexicon: "+ str(get_perc_manual(positive_words,negative_words,swn_positive,swn_negative))
print "Second automatic lexicon: "+ str(get_perc_manual(positive_words,negative_words,wv_positive,wv_negative))

#Calculate the accuracy of words in the automatic lexicon. Assuming that the manual lexicons are accurate, it calculates the percentage of words that occur in both positive and negative (respectively) lists of automatic and manual lexicons.
def get_lexicon_accuracy(manual_pos,manual_neg,auto_pos,auto_neg):
    common_words = set(manual_pos+manual_neg).intersection(set(auto_pos+auto_neg))-set(negative_seeds)-set(positive_seeds)
    return (len(set(manual_pos) & set(auto_pos) & common_words)+len(set(manual_neg) & set(auto_neg) & common_words))/len(common_words)*100

print "\nAccuracy of lexicons: "
print "First automatic lexicon: "+ str(get_lexicon_accuracy(positive_words,negative_words,swn_positive,swn_negative))
print "Second automatic lexicon: "+ str(get_lexicon_accuracy(positive_words,negative_words,wv_positive,wv_negative))
```

```
% of words in manual lexicons, also present in the automatic lexicon
First automatic lexicon: 7.42377375166
Second automatic lexicon: 37.7964354102

Accuracy of lexicons:
First automatic lexicon: 82.4701195219
Second automatic lexicon: 98.9415915327
```

The second lexicon shares the most common words with the manual lexicon, and has the most accurately classified words, as it uses the most intutive way of creative positive/negative lexicons i.e. by identifying the most similar words.

Lexicons for Classification

What if we used the lexicons for the main classification problem?

Let's create a function that calculates a polarity score for a sentence based on a given lexicon. We'll count the positive and negative words that appear in the tweet, and then return a +1 if there are more posiitve words, a -1 if there are more negative words, and a 0 otherwise.

We'll then compare the results of the three lexicons on the development set.

In [29]:
```
#All lexicons are converted to sets for faster preprocessing.
manual_pos_set = set(positive_words)
manual_neg_set = set(negative_words)

syn_pos_set = set(swn_positive)
syn_neg_set = set(swn_negative)

wordvec_pos_set = set(wv_positive)
wordvec_neg_set = set(wv_negative)

#Function to calculate the polarity score of a sentence based on the frequency of positive or negative words.
def get_polarity_score(sentence,pos_lexicon,neg_lexicon):
    pos_count = 0
    neg_count = 0
    for word in sentence:
        if word in pos_lexicon:
            pos_count+=1
        if word in neg_lexicon:
            neg_count+=1
    if pos_count>neg_count:
        return 1
    elif neg_count>pos_count:
        return -1
    else:
        return 0


#Function to calculate the score for each tweet, and compare it against the actual labels of the dataset and calculate/count the accuracy score.
def data_polarity_accuracy(dataset,datalabels,pos_lexicon,neg_lexicon):
    accuracy_count = 0
    for index,tweet in enumerate(dataset):
        if datalabels[index]==get_polarity_score([word for sentence in tweet for word in sentence],pos_lexicon,neg_lexicon):
            accuracy_count+=1
    return (accuracy_count/len(dataset))*100

print "Manual lexicon accuracy: "+str(data_polarity_accuracy(dev_data[0],dev_data[1],manual_pos_set,manual_neg_set))
print "First auto lexicon accuracy: "+str(data_polarity_accuracy(dev_data[0],dev_data[1],syn_pos_set,syn_neg_set))
print "Second auto lexicon accuracy: "+str(data_polarity_accuracy(dev_data[0],dev_data[1],wordvec_pos_set,wordvec_neg_set))
```

```
Manual lexicon accuracy: 45.2159650082
First auto lexicon accuracy: 38.9283761618
Second auto lexicon accuracy: 45.1612903226
```

As we can see, the results reflect the quality metric obtained from the previous section, with the manual and second lexicon (word vector) winning out, while still not being as good as a Machine Learning algorithm without the polarity information.

Polarity Lexicon with Machine Learning

To conclude, we'll investigate the effects of adding the polarity score as a feature for our statistical classifier.

We'll create a new version of our feature extraction function, to integrate the extra *feature* and retrain our logisitc regression classifier to see if there's an improvement.

In [30]:
```
def convert_to_feature_dicts_v2(tweets,manual,first,second,remove_stop_words,n):
    feature_dicts = []
    for tweet in tweets:
        # build feature dictionary for tweet
        feature_dict = {}
        if remove_stop_words:
            for segment in tweet:
                for token in segment:
                    if token not in stopwords and (n<=0 or total_train_bow[token]>=n):
                        feature_dict[token] = feature_dict.get(token,0) + 1
        else:
            for segment in tweet:
                for token in segment:
                    if n<=0 or total_train_bow[token]>=n:
                        feature_dict[token] = feature_dict.get(token,0) + 1
        if manual == True:
            feature_dict['manual_polarity'] = get_polarity_score([word for sentence in tweet for word in sentence],manual_pos_set,manual_neg_set)
        if first == True:
            feature_dict['synset_polarity'] = get_polarity_score([word for sentence in tweet for word in sentence],syn_pos_set,syn_neg_set)
        if second == True:
            feature_dict['wordvec_polarity'] = get_polarity_score([word for sentence in tweet for word in sentence],wordvec_pos_set,wordvec_neg_set)

        feature_dicts.append(feature_dict)
    return feature_dicts
```

In [33]:
```
training_set_v2 = convert_to_feature_dicts_v2(train_tweets,True,False,True,True,2)

training_data_v2 = vectorizer.fit_transform(training_set_v2)
```

In [34]:
```
dev_set_v2 = convert_to_feature_dicts_v2(dev_data[0],True,False,True,False,0)

development_data_v2 = vectorizer.transform(dev_set_v2)

log_clf_v2 = LogisticRegression(C=0.012,solver='lbfgs',multi_class='multinomial')

log_clf_v2.fit(training_data_v2,train_data[1])

log_predictions_v2 = log_clf_v2.predict(development_data_v2)

print "Logistic Regression V2 (with polarity scores) Accuracy: " + str(accuracy_score(dev_data[1],log_predictions_v2))
```

```
Logistic Regression V2 (with polarity scores) Accuracy: 0.507927829415
```

Though minimal, there was some improvement indeed in the classifier by integrating the polarity data.

This concludes our project of building a very basic 3-way polarity classifier for tweets.