Sentiment Analysis of IMBD Movie Reviews

Xin Zhang, Lu Zhai, Caiwei Wu

Brandeis LING131A Final Project

Background

Sentiment analysis is a challenging subject in machine learning. People express their emotions in language that is often obscured by sarcasm, ambiguity, and plays on words, all of which could be very misleading for both humans and computers.
In this project, we apply different machine learning models to predict sentiments of IMDB movie reviews.

Design and Implementation

Data Set

All data sets are located in `datasets/` folder. The labeled data set consists of 50,000 IMDB movie reviews, specially selected for sentiment analysis. The sentiment of reviews is binary, meaning the IMDB rating < 5 results in a sentiment score of 0, and rating >=7 have a sentiment score of 1. No individual movie has more than 30 reviews. The 25,000 review labeled training set does not include any of the same movies as the 25,000 review test set. In addition, there are another 50,000 IMDB reviews provided without any rating labels.

File descriptions

- **labeledTrainData** - The labeled training set. The file is tab-delimited and has a header row followed by 25,000 rows containing an id, sentiment, and text for each review.
- **testData** - The test set. The tab-delimited file has a header row followed by 25,000 rows containing an id and text for each review. Your task is to predict the sentiment for each one.
- **unlabeledTrainData** - An extra training set with no labels. The tab-delimited file has a header row followed by 50,000 rows containing an id and text for each review.
- **imdb_master.csv** - An extra training set to train deep learning model.

Data fields

- **id** - Unique ID of each review
- **sentiment** - Sentiment of the review; 1 for positive reviews and 0 for negative reviews
- **review** - Text of the review

Reading the Data

First, we want to read the tab-delimited file into Python. To do this, we can use the **pandas** package, which provides the `read_csv` function for easily reading and writing data files. You can install it by typing `pip install pandas` in terminal.

```
train = pd.read_csv("labeledTrainData.tsv", header=0, \
                    delimiter="\t", quoting=3)
```

Note that `header = 0` here means that the first row will be use as the column names, `delimiter = "\t"` means that the fields are separated by tabs and `quoting=3` (equivalent to `QUOTE_NONE`) will ignore doubled quotes in csv files.

The three columns of the data are "id", "sentiment", and "review.":

```
>>> train.columns.values
array([id, sentiment, review], dtype=object)
```

After reading the training set, we can take a look at a few reviews:

```
>>>print(train["review"][1])
```

> The film starts with a manager (Nicholas Bell) giving welcome investors (Robert Carradine) to Primal Park . A secret project mutating a primal animal using fossilized DNA, like ¨Jurassik Park¨, and some scientists resurrect one of nature's most fearsome predators, the Sabretooth tiger or Smilodon . Scientific ambition turns deadly, however, and when the high voltage fence is opened the creature escape and begins savagely stalking its prey - the human visitors , tourists and scientific.Meanwhile some youngsters enter in the restricted area of the security center and are attacked by a pack of large pre-historical animals which are deadlier and bigger . In addition , a security agent (Stacy Haiduk) and her mate (Brian Wimmer) fight hardly against the carnivorous Smilodons. The Sabretooths, themselves , of course, are the real star stars and they are astounding terrifyingly though not convincing. The giant animals savagely are stalking its prey and the group run afoul and fight against one nature's most fearsome predators. Furthermore a third Sabretooth more dangerous and slow stalks its victims.

> The movie delivers the goods with lots of blood and gore as beheading, hair-raising chills,full of scares when the Sabretooths appear with mediocre special effects.The story provides exciting and stirring entertainment but it results to be quite boring .The giant animals are majority made by computer generator and seem totally lousy .Middling performances though the players reacting appropriately to becoming food.Actors give vigorously physical performances dodging the beasts ,running,bound and leaps or dangling over walls . And it packs a ridiculous final deadly scene. No for small kids by realistic,gory and violent attack scenes . Other films about Sabretooths or Smilodon are the following : ¨Sabretooth(2002)¨by James R Hickox with Vanessa Angel, David Keith and John Rhys Davies and the much better ¨10.000 BC(2006)¨ by Roland Emmerich with with Steven Strait, Cliff Curtis and Camilla Belle. This motion picture filled with bloody moments is badly directed by George Miller and with no originality because takes too many elements from previous films. Miller is an Australian director usually working for television (Tidal wave, Journey to the center of the earth, and many others) and occasionally for cinema (The man from Snowy river, Zeus and Roxanne,Robinson Crusoe). Rating : Below average, bottom of barrel.

There are common issues such as punctuation, abbreviations and HTML tags such as "\
" when processing text from online. Next we will be cleaning the data.

Cleaning the Data

Removing HTML Markup

First, we'll remove the HTML tags using the **Beautiful Soup** package which could be installed by : `sudo pip install BeautifulSoup4`

```
1.  # Initialize the BeautifulSoup object on a movie review
2.  BeautifulSoupObject = BeautifulSoup(review, features="html.parser")
3.  # Calling get_text() gives the text of the review, without tags or mar
    kup
4.  review_text = BeautifulSoupObject.get_text()
```

Removing Punctuation, Numbers and Stopwords

Next, to remove punctuation and numbers, we will use **NLTK** packages and regular expressions with python built-in **re** library.

Stemming and Lemmatizing

We used `PorterStemmer` and created our own lemmatizer `my_lemmatizer`. First, we used data from treebank.tagged_sents to generate the mapping from 5000 most frequent words to their corresponding tags. Next, we created a Unigramtagger based on the map, where we back off to defult tagger "NN" if the word is not one of the map's keys. Then, we created our own lemmatizer based on the tagger we have written.

```
def my_lemmatizer(w):
    if tagger(w)[1] in ('NN', 'NNS', 'NNS$', 'NPS', 'NPS$'):
        if re.match(r'(.)+s$', w) != None:
            return w[:-1]
        elif re.match(r'(.)+(x|z|s|ch|sh)es$', w) != None:
            return w[:-2]
        elif re.match(r'(.)+men$', w) != None:
            return w[:-2] + 'an'
        elif re.match(r'(.)+ies$', w) != None:
            return w[:-3] + 'y'
        else:
            return w
    elif tagger(w)[1] in ('VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ'):
        if re.match(r'(.)+ies$', w) != None:
            return w[:-3] + 'y'
        elif re.match(r'(.)+e[ds]$', w) != None:
            return w[:-2]
        elif re.match(r'(.)+s$', w) != None:
            return w[:-1]
        elif re.match(r'(.)+ing$', w) != None:
            return w[:-3]
        else:
            return w
    elif tagger(w)[1] in ('JJ', 'JJS', 'JJR', 'JJT'):
        if re.match(r'(.)+er$', w) != None:
            return w[:-2]
        elif re.match(r'(.)+est$', w) != None:
            return w[:-3]
        else:
            return w
    else:
        return w
```

Creating Features from a Bag of Words

After cleaning the data and process the texts, we use **Bag of Words** to convert the data information we have to numeric representation for machine learning. The bag of words model collects all the words from all the documents, and models each by counting the number of times each word appears exemplified by the following two sentences:

> Sentence 1: “The dog loves the bones.”
>
> Sentence 2: “The cat loves the fishes, not the bones.”
>
> From these two sentences, our vocabulary is as follows:
>
> {the, dog, loves, bones, cat, fishes, not}
>
> To get our bags of words, we count the number of times each word occurs in each sentence.
>
> In Sentence 1, "the" appears twice, and "cat", "sat", "on", and "hat" each appear once, so the feature vector for Sentence 1 is:
>
> Sentence 1: {2, 1, 1, 1, 0, 0, 0}
>
> Similarlly, we can get:
>
> Feature vector of Sentence 2: {3, 0, 1, 0, 1, 1, 1}
>
> In the IMDB data, the vocabulary size is too large to collect all the words. Therefore, we only choose the 5000 most frequent words in the training data reviews.

We'll be using the `CountVectorizer` from **scikit-learn** which convert a collection of text documents to a matrix of token countsto create bag-of-words features. Here is an exmaple usage of `CountVectorizer` which clearly illustrates how we can create Bag of Words model from sentences.

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
        'This is the first document.',
        'This document is the second document.',
        'And this is the third one.',
        'Is this the first document?',
    ]
>>> vectorizer = CountVectorizer()
>>> X = vectorizer.fit_transform(corpus)
>>> print(vectorizer.get_feature_names())
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'th
is']
>>> print(X.toarray())
[[0 1 1 1 0 0 1 0 1]
```

```
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
```

In our implementation, we set most parameters to None because we have already cleaned the data in the previous steps and we will not be using CountVectorizer' own options to automatically preprocessing, tokenization, and stop word removal.

```
vectorizer = CountVectorizer(analyzer = "word", tokenizer = None,
preprocessor = None, stop_words = None, max_features = 5000)
train_data_features = vectorizer.fit_transform(list(train.clean_review
)).toarray()
test_data_features = vectorizer.transform(list(test.clean_review)).toa
rray()
```

After extracting features, we can check the results by:

```
>>> print(train_data_features.shape)
(25000, 5000)
```

This shows that the resulting feature vector has 25000 rows which indicates the number of reviews and 5000 columns which indicates the number of features(vocabulary).

After trainning the Bag of Words model, we can take a look at the most frequent vocabulary:

```
68475 thi
46744 movi
43343 film
43251 wa
26373 hi
24945 one
20509 like
15225 ha
14542 time
13742 good
13712 make
12733 charact
12733 get
12711 see
12653 veri
12542 watch
11913 stori
```

```
...
```

Now we have numeric training features from the Bag of Words and the original sentiment labels for each feature vector, so we are ale to do some supervised learning! First, we'll use the Random Forest classifier in **scikit-learn**.

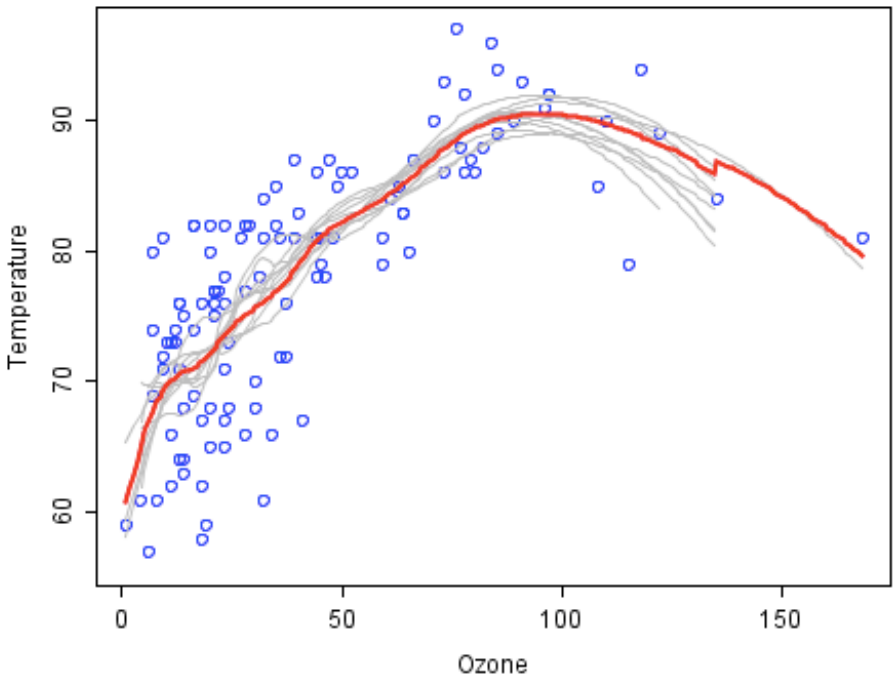
Random Forest

Introduction of the algorithm

an Ensemble Method

The random forest is an ensemble approach that can also be thought of as a form of nearest neighbor predictor. Ensembles are a divide-and-conquer approach used to improve performance. The main principle behind ensemble methods is that a group of "weak learners" can come together to form a "strong learner". The figure below (taken from here) provides an example. Each classifier, individually, is a "weak learner," while all the classifiers taken together are a "strong learner".

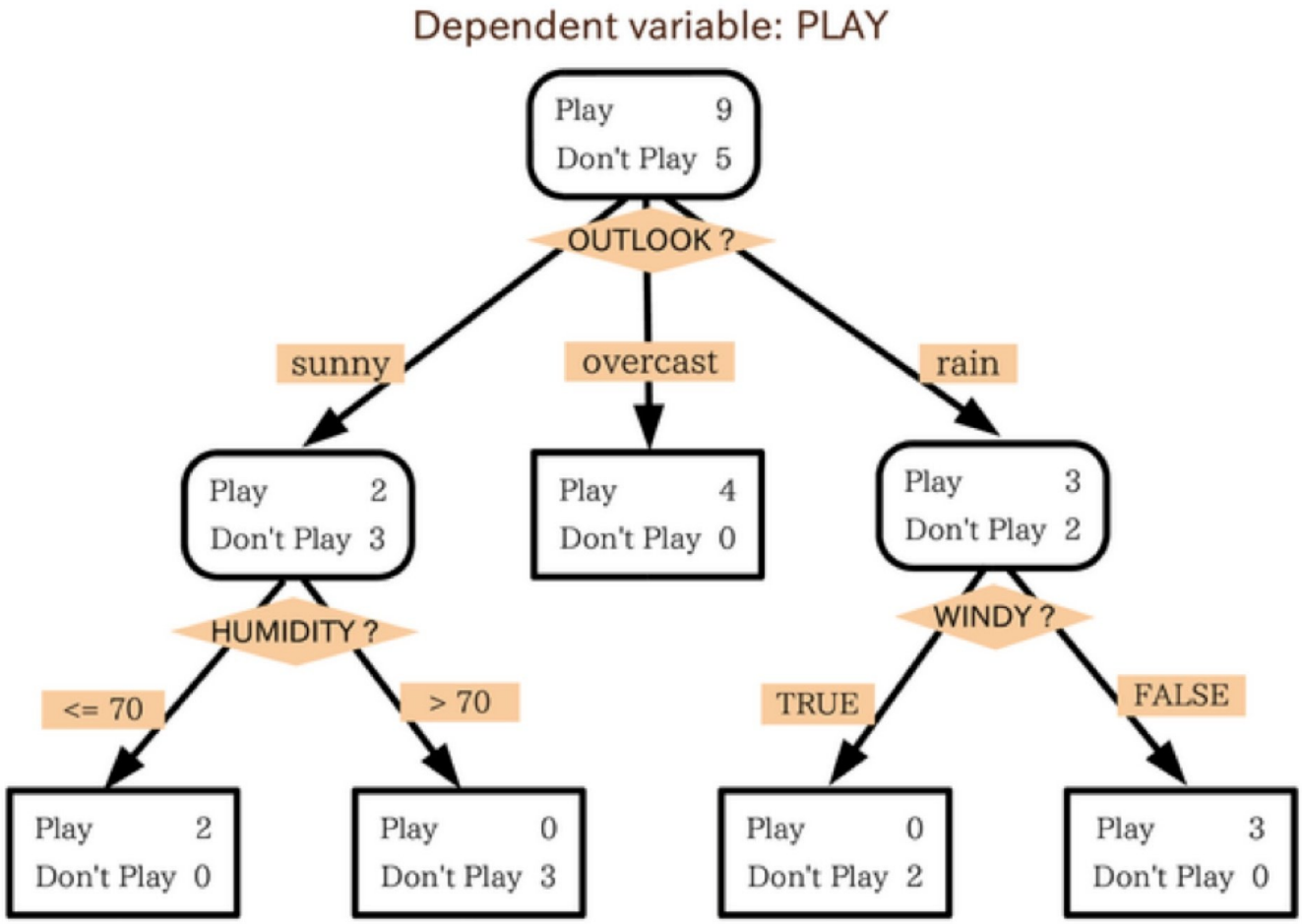
The data to be modeled are the blue circles. We assume that they represent some underlying function plus noise. Each individual learner is shown as a gray curve. Each gray curve (a weak learner) is a fair approximation to the underlying data. The red curve (the ensemble "strong learner") can be seen to be a much better approximation to the underlying data.



Trees and Forests

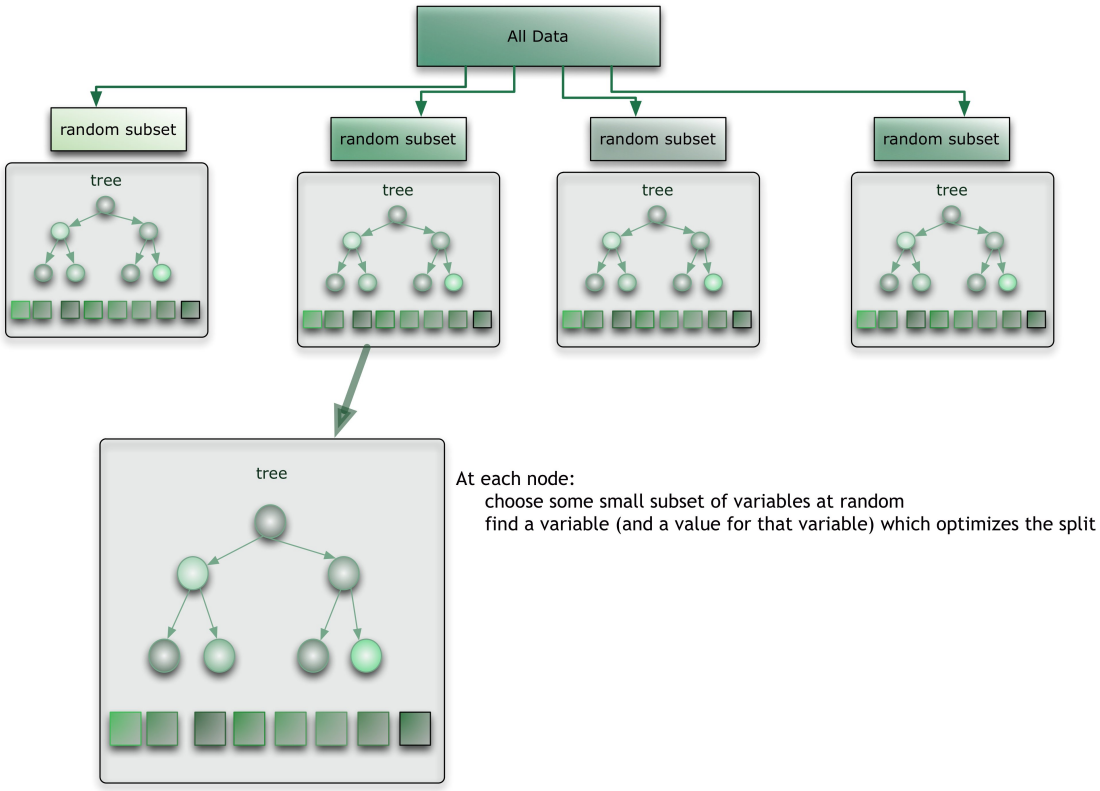
The random forest starts with a standard machine learning technique called a "decision tree" which, in ensemble terms, corresponds to our weak learner. In a decision tree, an input is entered at the top and as it traverses down the tree the data gets bucketed into smaller and smaller sets.

In the following example, the tree advises us, based upon weather conditions, whether to play ball. For example, if the outlook is sunny and the humidity is less than or equal to 70, then it’ s probably OK to play.



The random forest takes this notion to the next level by combining trees with the notion of an ensemble. Thus, in ensemble terms, the trees are weak learners and the random forest is a strong learner.

-



Here is how such a system is trained; for some number of trees T:

Sample N cases at random with replacement to create a subset of the data (see top layer of figure above). The subset should be about 66% of the total set.
At each node:
For some number m (see below), m predictor variables are selected at random from all the predictor variables.
The predictor variable that provides the best split, according to some objective function, is used to do a binary split on that node.
At the next node, choose another m variables at random from all predictor variables and do the same.
Depending upon the value of m, there are three slightly different systems:

- Random splitter selection: m =1
- Breiman' s bagger: m = total number of predictor variables
- Random forest: m << number of predictor variables. Brieman suggests three possible values for m: ½√m, √m, and 2√m

Running a Random Forest

When a new input is entered into the system, it is run down all of the trees. The result may either be an average or weighted average of all of the terminal nodes that are reached, or, in the case of categorical variables, a voting majority.

Note that:

- With a large number of predictors, the eligible predictor set will be quite different from node to node.
- The greater the inter-tree correlation, the greater the random forest error rate, so one pressure on the model is to have the trees as uncorrelated as possible.
- As m goes down, both inter-tree correlation and the strength of individual trees go down. So some optimal value of m must be discovered.

Here, we set the number of trees to 100 as a reasonable default value. More trees may (or may not) perform better, but will certainly take longer to run. Likewise, the more features you include for each review, the longer this will take.
The accuracy we get for this approach were aorund 0.84-0.85.

```
forest = RandomForestClassifier(n_estimators = 100)
# Fit the forest to the training set, using the bag of words as
# features and the sentiment labels as the response variable
forest = forest.fit( train_data_features, train["sentiment"] )
```

Naïve Bayes

In machine learning, Naïve Bayes classifiers are a family of classifiers that use Bayes' s theorem to determine the probability of a certain class based on the probability of the feature set associate with this class. The theorem is:

$$P(C|F) = \frac{P(F|C) \times P(C)}{P(F)}$$

- P(C|F) = the probability of class C given the feature set F
- P(F|C) = the probability of the feature set F given class C
- P(C) = the probability of the class C
- P(F) = the probability of the feature set F

The Naïve Bayes extends the above bayes' theorem to all the features. It considers each feature in the set as an independent data point. The formula looks like this:

$$P(C|F) = \frac{(F_1|C) \times P(F_2|C) \times \ldots \times P(F_n|C) \times P(C)}{(F_1) \times P(F_2) \times \ldots P(F_n)}$$

Since naïve bayes is one of the most common algorithm in machine learning, many package have build in function to do the prediction. Here, we use basic mathematic calculation method, nltk Naïve Bayes classifier, and Scikit-learn to create bag-of-words features. A total of 25000 labeled reviews were used to test the accuracy of each method. Within the 25000 review, 90% were used as training data, and 10% were used as testing data

Basic mathematics calculation:
we do the calculation according to the following steps:

1. Use the training data sets to find the total number of positive reviews, total number of negative reviews, and total number of reviews.

   P(C(positive)) = total number of positive reviews/total number of reviews

   P(C(negative)) = total number of positive reviews/total number of reviews
2. Collect all the words exits in the positive reviews and negative reviews, respectively.

   For each word, calculate how many times it occurs in positive reviews and negative reviews (occurrence_in_class).

   Also calculate total number of words in each classes (total_words_in_class).
3. Do the prediction:

   For each word in the given review text, calculate

   1. How many times it occurs in the given review text (occurrence_in_text).
   2. Find it occurrence in given class generated in step 2 (occurrence_in_class). If it is not exit in given class dictionary, which is generated using training data set, set the occurrence as 0.
   3. P(F|C) = occurrence_in_text * (occurrence_in_class +1)/(total_words_in_class)

      Add is to smooth the value, so we won't multiply the occurrence by 0 if we have word that doesn't exit in the training data.
   4. P(C(pos)|F) = P(F1|C(pos)) *P(F2|C(pos))*...P(Fn|C(pos))*P(C(pos)/P(F)

      P(C(neg)|F) = P(F1|C(neg)) *P(F2|C(neg))*...P(Fn|C(neg))*P(C(neg)/P(F)

      We can ignore the denominator P(F) because it is identical in both classes.

      If P(C(pos)|F)> P(C(neg)|F), we do the positive prediction, otherwise, we do the negative prediction
4. Calculate the accuracy by compare the prediction result with the original sentiment label.

   Accuracy = the number of reviews, whose prediction result is the same as its original sentiment label /total number of test reviews.

Using this method, the accuracy is 0.712.

NLTK Naïve Bayes classifier

NLTK contain NaiveBayesClassifier, therefore, we don't need to calculate the probability by our self. We can just create feature sets and use the NaiveBayesClassifier to the prediction. In order to use NLTK Naïve Bayes classifier, we need to build our own feature sets. Here we will use the concept of Bag of Words, which has been described before. Briefly, we choose the 5000 most frequent words in the training data reviews, and used these words as features. Then, we get the feature sets by calculating the occurrence of each of the 5000 words in each review in the training data and testing data. The training feature sets were used to to train the classifier, and the testing feature sets were used to do the evaluation.
The accuracy we got is 0.8476.

Scikit-learn

Scikit-learn is a python machine learning library, which could be used to create bag-of-word features. This is the easiest way to get the feature sets. We just need to input all the "clean_review" into the CountVectorizer, which is scikit-learn's bag of words tool. The Countvectorizer will fits the model and learn the vocabulary, then transform the training data into feature vectors.

There are multiple Naïve Bayes classifiers in Scikit-learn. Here we choose the 3 most common classifiers, Multinomial, Bernoulli, and Gaussian Naïve Bayes classifiers to do the sentiment analysis.

Multinomial Naïve Bayes: The multinomial is useful when the feature is discrete counts. For example, the counts of word occur in the text. The counts are discrete. Therefore, this model fit our features set.

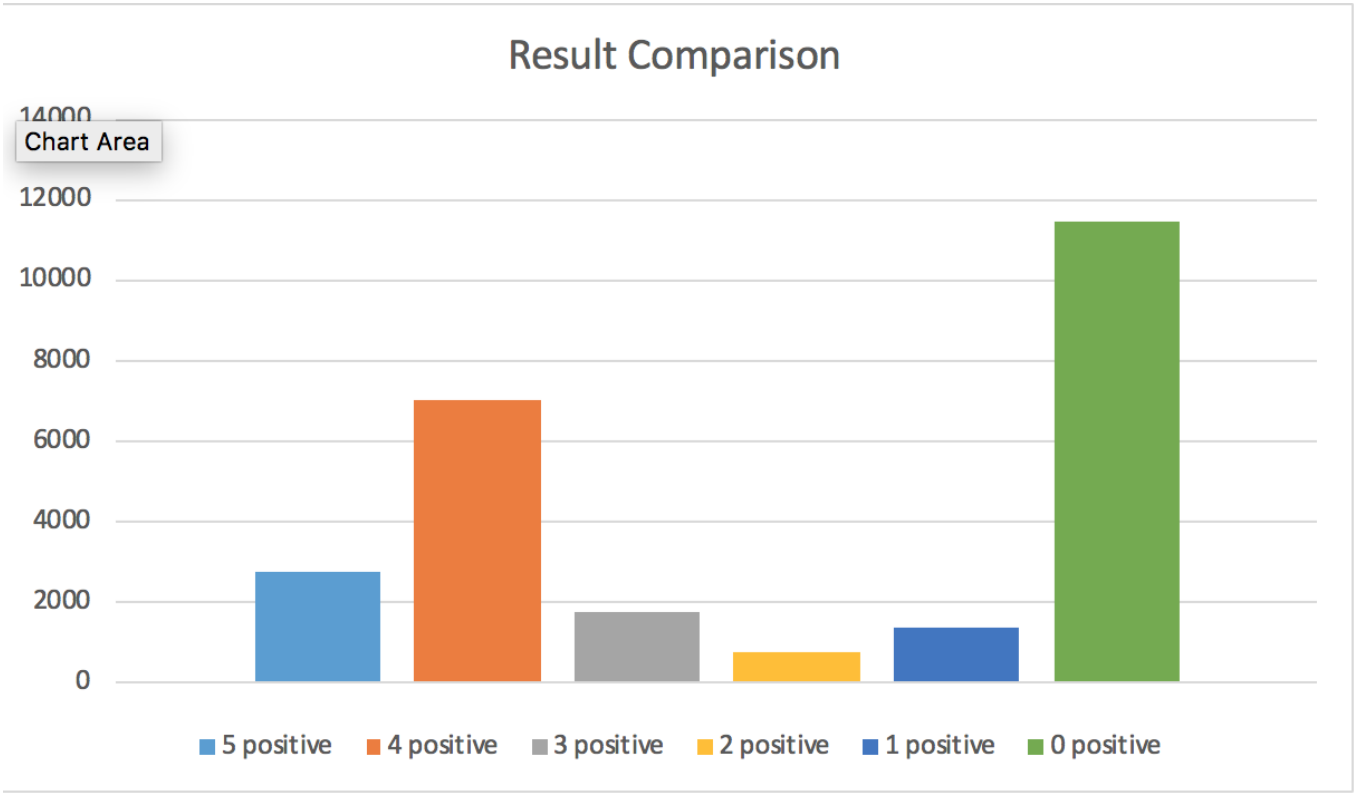
Bernoulli Naïve Bayes: The Bernoulli model is useful when the feature is binary (i.e. 1 or 0, yes or no…). It could be used in a bag of words model, where 0 stands for word does not occur in the text and 1 stands for word occurs in the text. In our feature set, some words only exist once in the review, Therefore, this model may fit our features set partially.

Gaussian Naïve Bayes: The Gaussian model is useful when the features are continuous. (e.g., the student dataset where features are height, weight). Therefore, it may not fit our features sets.

The accuracy we got from multinomial naïve bayes is 0.8496, from Bernoulli naïve bayes is 0.8512, and from Gaussian naïve bayes is 0.7312. This result tally with the model analysis that both multinomial and Bernoulli model fit our data sets, but Gaussian naïve bayes model doesn't fit very well.

Prediction of testData.tsv:

We have 25000 unlabeled review data in the testData.tsv, and our aim is to do the sentiment analysis of these unlabeled data. All the 25000 labeled review data in the labeledTrainData.tsv were used as training set to do the classifier training. Each method described above gave us a prediction result. All the results were summarized in the table:



There are:

- 2735 reviews are identified as positive in all of the 5 algorithms.
- 7007 reviews are identified as positive in 4 of the 5 algorithms.
- 1746 reviews are identified as positive in 3 of the 5 algorithms.
- 742 reviews are identified as positive in 2 of the 5 algorithms.
- 1340 reviews are identified as positive in 1 of the algorithms.
- 11430 reviews are identified as negative in all of the 5 algorithms.

The results show that all the algorithms can generate more identical results for the negative reviews.

Word2Vec

Introducing Distributed Word Vectors

Word2vec, published by Google in 2013, is a neural network implementation that learns distributed representations for words. Other deep or recurrent neural network architectures had been proposed for learning word representations prior to this, but the major problem with these was the long time required to train the models. Word2vec learns quickly relative to other models.

How does it work

Word2Vec does not need labels in order to create meaningful representations. This is useful, since most data in the real world is unlabeled. If the network is given enough training data (tens of billions of words), it produces word vectors with intriguing characteristics.

Word2Vec takes a text corpus as input and produces the word vectors as output. It first constructs a vocabulary from the training text data and then learns vector representation of words. The resulting word vector file can be used as features in many natural language processing and machine learning applications.

A simple way to investigate the learned representations is to find the closest words for a user-specified word. For example, we can find the similar words as 'france' and their distances to 'france', which should look like:

Word Cosine distance

```
            spain              0.678515
          belgium              0.665923
      netherlands              0.652428
            italy              0.633130
      switzerland              0.622323
       luxembourg              0.610033
         portugal              0.577154
           russia              0.571507
          germany              0.563291
        catalonia              0.534176
```

There are two main learning algorithms in Word2Vec : continuous bag-of-words and continuous skip-gram. Both algorithms learn the representation of a word that is useful for prediction of other words in the sentence. These algorithms are described in detail in [1,2].

Interesting properties of the word vectors

Word2Vec does not need labels in order to create meaningful representations. This is useful, since most data in the real world is unlabeled. If the network is given enough training data (tens of billions of words), it produces word vectors with intriguing characteristics. Words with similar meanings appear in clusters, and clusters are spaced such that some word relationships, such as analogies, can be reproduced using vector math. The famous example is that, with highly trained word vectors, "king - man + woman = queen."

From words to phrases and beyond

In certain applications, it is useful to have vector representation of larger pieces of text. This can be achieved by `gensim.models.phrases` module which lets you automatically detect phrases longer than one word. Using phrases, you can learn a word2vec model where "words" are actually multiword expressions, such as san_francisco or financial_crisis.

The example output with the closest tokens to 'san_francisco' looks like:

Word Cosine distance

```
       los_angeles             0.666175
       golden_gate             0.571522
           oakland             0.557521
        california             0.554623
         san_diego             0.534939
          pasadena             0.519115
           seattle             0.512098
             taiko             0.507570
           houston             0.499762
  chicago_illinois             0.491598
```

The linearity of the vector operations seems to weakly hold also for the addition of several vectors, so it is possible to add several word or phrase vectors to form representation of short sentences

Using word2vec in Python

In Python, we will use the excellent implementation of word2vec from the **gensim** package.
Although **Word2Vec** does not require graphics processing units (GPUs) like many deep learning algorithms, it is compute intensive. Both Google's version and the Python version rely on multi-threading (running multiple processes in parallel on your computer to save time). In order to train our model in a reasonable amount of time, we need to install **cython**. Word2Vec will run without cython installed, but it will take days to run instead of minutes.

Preparation before training the model

First, we read in the data with pandas, note that we now use unlabeledTrain.tsv, which contains 50,000 additional reviews with no labels. When we built the Bag of Words model previously, extra unlabeled training reviews were not useful. However, since Word2Vec can learn from unlabeled data, these extra 50,000 reviews can now be used.

The functions we write to clean the data are also similar to previous ones, although now there are a couple of differences. First, to train Word2Vec it is better not to remove stop words because the algorithm relies on the broader context of the sentence in order to produce high-quality word vectors. For this reason, we will make stop word removal optional here.

Next, we want a specific input format. Word2Vec expects single sentences, each one as a list of words. In other words, the input format is a list of lists.

It is not at all straightforward how to split a paragraph into sentences. There are all kinds of gotchas in natural language. English sentences can end with "?", "!", """, or ".", among other things, and spacing and capitalization are not reliable guides either. For this reason, we'll use NLTK's punkt tokenizer for sentence splitting.

```
def prepare_sentences():
    train, test, unlabeled_train = read_data()
    tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
    sentences = []
    print("Parsing sentences from training set")
    for review in train["review"]:
        sentences += review_to_sentences(review, tokenizer)
    print("Parsing sentences from unlabeled set")
    for review in unlabeled_train["review"]:
        sentences += review_to_sentences(review, tokenizer)
    return sentences
```

Training the model

After getting all the input sentences with expected format, we're ready to train the model. There are a number of parameter choices that affect the run time and the quality of the final model that is produced. Some of the most important ones are as follows:

- **Architecture**: Architecture options are skip-gram (default) or continuous bag of words. We found that skip-gram was very slightly slower but produced better results.
- **Training algorithm**: Hierarchical softmax (default) or negative sampling. For us, the default worked well.
- **Downsampling of frequent words**: The Google documentation recommends values between .00001 and .001. For us, values closer 0.001 seemed to improve the accuracy of the final model.
- **Word vector dimensionality**: More features result in longer runtimes, and often, but not always, result in better models. Reasonable values can be in the tens to hundreds; we used 300.

- **Context / window size** : How many words of context should the training algorithm take into account? 10 seems to work well for hierarchical softmax (more is better, up to a point).
- **Worker threads**: Number of parallel processes to run. This is computer-specific, but between 4 and 6 should work on most systems.
- **Minimum word count**: This helps limit the size of the vocabulary to meaningful words. Any word that does not occur at least this many times across all documents is ignored. Reasonable values could be between 10 and 100. In this case, since each movie occurs 30 times, we set the minimum word count to 40, to avoid attaching too much importance to individual movie titles. This resulted in an overall vocabulary size of around 15,000 words. Higher values also help limit run time.

Have some fun with the model

Let's explore the model we created out of our 75,000 training reviews.

Our model is capable of distinguishing differences in meaning! It knows that men, women and children are more similar to each other than they are to kitchens. More exploration shows that the model is sensitive to more subtle differences in meaning, such as differences between countries and cities:

```
>>> model.wv.doesnt_match("man woman child kitchen".split())
kitchen
>>> model.wv.doesnt_match("france england germany berlin".split())
berlin
```

We can also use the "most_similar" function to get insight into the model's word clusters:

```
>>> model.wv.most_similar('wonderful')
[('fantastic', 0.7783489227294922), ('marvelous', 0.7624002695083618),
('superb', 0.7221430540084839), ('great', 0.7213148474693298),
('terrific', 0.720775842666626), ('brilliant', 0.7028141021728516),
('magnificent', 0.6984641551971436), ('delightful', 0.6961362957954407
),
('splendid', 0.6876288056373596), ('fabulous', 0.6778663992881775)]

>>>model.wv.most_similar('terrible')
[('horrible', 0.8879971504211426), ('horrid', 0.7927792072296143),
('horrendous', 0.7827708721160889), ('atrocious', 0.7745435237884521),
('awful', 0.7733446359634399), ('dreadful', 0.7665877938270569),
('lousy', 0.7277438044548035), ('abysmal', 0.6998542547225952),
('bad', 0.6812251806259155), ('appalling', 0.6735174655914307)]
```

Next we will explore how we can use these fancy distributed word vectors for supervised learning.

Numeric Representations of Words

If we take a closer look, we can find out that the Word2Vec model trained consists of a feature vector for each word in the vocabulary, stored in a numpy array called "syn0":

```
>>> # Load the model that we created
>>> from gensim.models import Word2Vec
>>> model = Word2Vec.load("300features_10context")

>>> type(model.wv.vectors)
<type 'numpy.ndarray'>

>>> model.wv.vectors.shape
(16490, 300)
```

The number of rows in vectors is the number of words in the model's vocabulary, and the number of columns corresponds to the size of the feature vector, which was set by us. Setting the minimum word count to 40 gave us a total vocabulary of 16,490 words with 300 features apiece. Individual word vectors can be accessed in the following way:

From Words To Paragraphs, Experiment One: Vector Averaging

One challenge with the IMDB dataset is the variable-length reviews. We need to find a way to take individual word vectors and transform them into a feature set that is the same length for every review.

Since each word is a vector in 300-dimensional space, we can use vector operations to combine the words in each review. One method we tried was to simply average the word vectors in a given review (for this purpose, we removed stop words, which would just add noise).

The accuracy we get for this part were just a little better than chance, we wanted to find a better approach.

Name	Submitted	Wait time	Execution time	Score
Word2Vec_AverageVectors.csv	a day ago	0 seconds	0 seconds	0.50012

Complete

From Words to Paragraphs, Experiment Two: Clustering

The word vectors can be also used for deriving word classes from huge data sets. This is achieved by performing **K-means** clustering on top of the word vectors. Grouping vectors in this way is known as "vector quantization." To accomplish this, we first need to find the centers of the word clusters, which we can do by using a clustering algorithm such as **K-Means**.
The output is a vocabulary file with words and their corresponding class IDs, such as:

```
carnivores 234 carnivorous 234 cetaceans 234 cormorant 234 coyotes 234
crocodile 234 crocodiles 234 crustaceans 234 cultivated 234 danios 234
```

```
. . .
acceptance 412 argue 412 argues 412 arguing 412
argument 412 arguments 412 belief 412 believe 412 challenge 412 claim 4
12
```

In **K-Means**, the one parameter we need to set is "K," or the number of clusters. How should we decide how many clusters to create? Trial and error suggested that small clusters, with an average of only 5 words or so per cluster, gave better results than large clusters with many words. Clustering code is given below. We use **scikit-learn** to perform our K-Means.

```
from sklearn.cluster import KMeans

    start = time.time()
    # Set "k" (num_clusters) to be 1/5th of the vocabulary size, or an
    # average of 5 words per cluster
    word_vectors = model.wv.vectors
    num_clusters = int(word_vectors.shape[0] / 5)

    # Initalize a k-means object and use it to extract centroids
    kmeans_clustering = KMeans(n_clusters=num_clusters)
    idx = kmeans_clustering.fit_predict(word_vectors)

    # Get the end time and print how long the process took
    end = time.time()
    elapsed = end - start
    print("Time taken for K Means clustering: ", elapsed, "seconds.")
```

The cluster assignment for each word is now stored in `idx`, and the vocabulary from our original `Word2Vec` model is still stored in `model.wv.index2word`. For convenience, we zip these into one dictionary as follows:

```
# Create a Word / Index dictionary, mapping each vocabulary word
# to a cluster number
word_centroid_map = dict(zip(model.wv.index2word, idx))
```

We are eager to see what exactly are in the clusters since it might sounds abstract, note that the results could differ since Word2Vec relies on a random number seed. Here is a loop that prints out the words for clusters 0 through 9:

```
Cluster 0
['riff', 'franz', 'corbett', 'mastroianni', 'heath', 'edwin', 'pusher',
'ledger', 'ji', 'villainy']

Cluster 1
```

```
['defies', 'transcends', 'subjective']

Cluster 2
['rescue', 'transport', 'supplies', 'retrieve', 'repair', 'hatch']

Cluster 3
['corporate', 'corruption', 'terrorism', 'organized', 'espionage', 'dec
eption',
'procedure', 'triad',
'deceit', 'conspiracies', 'morale']

Cluster 4
['introduction', 'selection', 'inclusion']

Cluster 5
['theaters', 'uk', 'cinemas', 'theatres']

Cluster 6
['tentacles']

Cluster 7
['merchant', 'graf', 'ivory', 'das', 'battleship']

Cluster 8
['dry', 'cheeky', 'cheery', 'bawdy']

Cluster 9
['fanatical']
```

We can see that the clusters are of varying quality. Cluster 5 mostly contains words that have similar meaning to "theater", and Cluster 2 contains mostly verbs. On the other hand, cluster 8 is a little bit mystifying, what does "cheery" and "dry" have in common?

At least now we have a cluster (or "centroid") assignment for each word, and we can convert reviews into **bags-of-centroids**. This works just like **Bag of Words** but uses semantically related clusters instead of individual words:

```
def bag_of_centroids_for_a_review(wordlist, word_centroid_map):
    # The number of clusters is equal to the highest cluster index
    # in the word / centroid map
    num_centroids = max(word_centroid_map.values()) + 1
    bag_of_centroids = np.zeros(num_centroids, dtype="float32")
    # Loop over the words in the review. If the word is in the
vocabulary,
    # find which cluster it belongs to, and increment that cluster
```

```
    count
8.      # by one
9.      for word in wordlist:
10.         if word in word_centroid_map:
11.             index = word_centroid_map[word]
12.             bag_of_centroids[index] += 1
13.     return bag_of_centroids
```

The function above will give us a numpy array for each review, each with a number of features equal to the number of clusters. Finally, we create bags of centroids for our training and test set, then train a random forest and extract results. The results has been saved to file `BagOfCentroids.csv`. We found that this approach gives about the same (or slightly worse) results compared to the Bag of Words model.

TF-IDF and SVM

TF-IDF

Tf means term-frequency while tf-idf means term-frequency times inverse document-frequency. This is a common term weighting scheme in information retrieval, that has also found good use in document classification.

The goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus.

The data process part remain the same as Word2Vec, and we combined CountVectorizer and TfidfTransformer to get a normalized tf-idf representation of the words.

```
1.  vectorizer = CountVectorizer(analyzer='word', max_features=2500)
2.
3.      train_data_features = vectorizer.fit_transform(clean_train_reviews)
    # Vectorizing training Data
4.      train_data_features = train_data_features.toarray()
5.
6.      test_data_features = vectorizer.transform(clean_test_reviews)   # Ve
    ctorize Test Data
7.      test_data_features = test_data_features.toarray()
8.
9.      tfidf_transformer = TfidfTransformer().fit(train_data_features)   #
    TFIDF
10.     messages_tfidf = tfidf_transformer.transform(train_data_features)
```

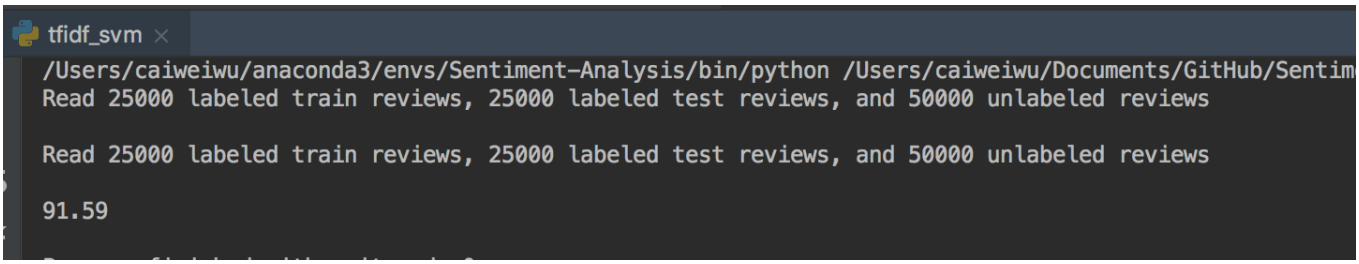
```
11.         test_tfidf = tfidf_transformer.transform(test_data_features)
```

Linear Support Vector Classification

Linear Support Vector Classification is Similar to SVC with parameter kernel=’ linear’ , but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

This model has reached the accuracy of above 0.91.



Deep Learning Model

We also played with deep learning models such as **word embedding**, **LSTM** and **CNN**. We used **Keras** and **tensorflow** library for these functions. Since larger data set produces better results for most deep learning methods, we added one more dataset `imdb_master.csv` for trainning our deep learning model.
The data processing part is similar to previous ones except that we dropped unecessary columns from additional dataset and combine the original datasets and the new dataset as one. Here we stacked mutiple models by:

- Create word2vec dictionary and word embbeding
- Apply CNN with maxpooling to find out features of negative and positive sentiment
- Apply LSTM bi directional unit to for need of good memory

The accuracy for this model is around 0.91- 0.92.
Note:

- **tensorflow** has trouble running in Python 3.7, so we run it in **Python 3.5**. Any other models in our projects runs perfectly fine in **Python 3.7 **
- It takes quite long to run this model, about 20mins on a Mac-pro with 2.6 GHz Intel Core i7 processor and 32GM Memory.

References

1. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In Proceedings of Workshop at ICLR, 2013.
2. Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of NIPS, 2013.
3. http://blog.citizennet.com/blog/2012/11/10/random-forests-ensembles-and-performance-metrics
4. https://www.kaggle.com/c/word2vec-nlp-tutorial