Lexicon Generation for Sentiment Classification (Ch 18 3E)

We can generate labeled lexicons to aid in certain tasks, such as sentiment analysis. We can train a classifier to focus on known positive and negative words in order to determine if a given text is positive or negative. Sources like SentiWordNet provide a large lexicon with sentiment-based annotation, but how can we extract even more words without manually sifting through the dictionary? This is where semi-supervised induction of sentiment lexicons comes in. Basically, we are bootstrapping the generation of more sentiment-labeled words using an existing annotated lexicon. One method to achieve this is through the use of seed words and adjective coordination.

Step 0: Overview and Tutorial Set up

As mentioned above, we will be using a semi supervised learning method to find words that have positive or negative meanings. In short, we will start with a human labeled list of positive and negative words. We will then search a corpus for conjunctions that include one of the seed words. For example, an “and” conjunction such as “happy and excited” probably means that the words “happy” and “excited” have similar meanings. If we know that “happy” is a positive word, then we can guess that “excited is also a positive word. Similarly, a conjunction with the word “but” usually implies opposite meanings. If we see the conjunction “happy but stressed,” we can probably learn that “stressed” is a negative word. With enough data, we can classify unseen words as positive or negative, depending on how often it appears with our seed words. This tutorial will go through the process of classifying unseen words as positive or negative, and then a comparison of how the accuracy of sentiment classification is affected by adding more learned words to the lexicon.

The first step is to this task is to grab the source files. The easiest way to do this is to clone the git repository located at the following url:
`https://github.com/Hemphill39/nlp-final-tutorial.git`

We’ll also need to install some packages. First change your directory to the one you just cloned, then run:

```
> pip install -r requirements.txt
```

To install the required packages. Additionally, if you want to play with all of the files, you may need to install some corpora from nltk. To do so, run the script corpus_setup.py. This will download additional data required by nltk.

There are a couple files we should go over before starting the tutorial. To simplify the focus of the tutorial, we have already gone through the process of finding seed words and and/but conjunctions. Your job will be to construct a distance matrix between unseen words and seed words. After that, you will run a sentiment classifier with and without the unseen words and for a comparison of the accuracy. The seed words, “and” relationships, and “but” relationships are located in the files “pos_words.txt”, “neg_words.txt”, “and_relationships.txt”, and “but_relationships.txt.” Additionally, the code that was used to construct these files is located in “seed_words.py”, “and_but_relationships.py”, and “tutorial_util.py.” These files won’t be directly

used, but they are provided for reference and will be discussed later. The majority of this tutorial will take place in “polarity_graph.py” and “rateLexicon.py”

Step 1: Preliminary Rating

First let's see how our classifier does with our existing seed words. In the folder provided to you, there is a python script called rateLexicon.py. This script runs through a series of reviews, filtering out any words which do not appear within the seed lexicons, and then trains and tests a multinomial naive bayes classifier on it. Run this classifier on the existing positive and negative seed words by entering the following in your command line:

```
> python3 rateLexicon.py pos_words.txt neg_words.txt
```

You should get an accuracy of about 80%

Step 2: Create Seed Lexicon

Now the fun starts. Step 2 is to create a seed lexicon. The process is discussed here for reference and the code for creating a seed lexicon can be found in “seed_words.py.”

For this tutorial, we are using SentiWordNet sentiment labeled word sets to create a lexicon. Each word in SentiWordNet is assigned a positive score, negative score, and objective score, all on a scale from 0 to 1. To create a list of positive and negative words, we find all of the words that have a very positive or a very negative score. For this tutorial we used a threshold of 0.7, which gives us about 500 positive and 500 negative words for our lexicon. These words are stored in the files “pos_words.txt” and “neg_words.txt.”

Step 3: Find cues to candidate similar words

Step 3 has already been completed for you as well, but an explanation of the code in “and_but_relationships.py” follows. We want to find all uses of the seed words in “and” and “but” conjunctions. To complete this task, we will use the *Reviews* and *Brown* corpus that is included in the nltk python packages. We first construct a Text object from the words of the above corpus. Nltk also provides a powerful search tool that will return a list of tokens based on a regular expression. Let’s create a text object from a corpus, and then a token search based on this text.

```
# get our imports out of the way
from nltk.corpus import brown, movie_reviews
from nltk.text import TokenSearcher
```

```
# create a text object of all the words in the movie reviews corpus
reviews = nltk.Text(movie_reviews.words())
# create a token search to search the text
tok_search = TokenSearcher(text)
```

Now that we have a token searcher, we need to think of how we want to find word sequences that match our conjunctions. Below is a code snippet that uses regular expressions to find these matches, and returns a list of lists that includes the tokens that match.

```
and_sequences = tok_search.findall('<\w*> <and> <\w*>')
but_sequences = tok_search.findall('<\w*> <but> <\w*>')
```

The above code snippets will find all of the tokens that match the regular expression `'<\w*> <and> <\w*>'`. The regular expression will match a single word, <\w*>, a space, the word "and", a space, and then another word, <\w*>. After all of the matching sequences are found, we can write the matching words to its respective file. We write all of the word pairs that include "and" to the "and_relationships.txt" file, and all of the word pairs that include "but" to the "but_relationships.txt" file. We will use these files in the next, and most exciting step, the polarity graph.

Step 4: Create Distance Matrix

The most important part of this tutorial is constructing the distance matrix, which will be used to classify words as positive or negative. Each word that needs to be classified will have a distance vector, where each entry in the vector is the number of positive associations with a seed word. Combining all of these vectors together will create a matrix whose columns are the seed words, and whose rows are the distance vector for a word we want to classify. Each cell in the matrix represents the relationship between the word and the seed word in that column. If the cell has a negative value, it is a negative word, if it is a positive value, it is a positive word, at least in this distance measure.

The idea of populating the matrix is simple. First, we will load the "and" and "but" relationships from their files. Then, we can start iterating through the relationships, keeping track of which new words occur with which seed words. If the new word appears in an "and" conjunction and the seed word is positive, increase the score by 1. If the seed word is negative, decrease the score by one. The opposite is true for "but" conjunctions.

To complete this task in python, we are going to create a dictionary where each key is a word that needs to be classified, and each value is another dictionary. In the nested dictionary, each

key will be a seed word and each value will be the score between the new word and the seed word.

```
# Let’s start by importing some libraries and reading in the conjunctions.
from tutorial_util import load_relationships, load_seed_words

pos_words, neg_words = load_seed_words('pos_words.txt', 'neg_words.txt')
and_relationships = load_relationships('and_relationships.txt')
but_relationships = load_relationships('but_relationships.txt')

distance_matrix = {}
```

`and_relationships` and `but_relationships` are dictionaries whose keys are the new words and whose values are a list of seed words that they occur with. Next, we can iterate across the relationships and appropriately adjust the score in the dictionaries.

```
for new_word in and_relationships:
   # initialize the nested dictionary if not already
   if new_word not in distance_matrix:
       distance_matrix[new_word] = {}

   # adjust the score appropriately, increasing the score for positive words
   # And decreasing scores for negative words
   for seed_word in and_relationships[new_word]:
       # initialize the counter in the nested dictionary
       if seed_word not in distance_matrix[new_word]:
           distance_matrix[new_word][seed_word] = 0

       # adjust the score accordingly. Do the opposite of this for but
       #relationships
       if seed_word in pos_words:
           distance_matrix[new_word][seed_word] += 1

       if seed_word in neg_words:
           distance_matrix[new_word][seed_word] -= 1
```

After writing similar code for “but” relationships, we now have a distance matrix that we can use with any type of clustering or graph algorithms. We will utilize a bayes classifier because it is a very well known algorithm and will be easy to compare results.

Step 5: Cluster/Classify the graph
Now that we’ve populated the distance matrix, we need to use it to classify each new word as positive, negative, or neutral and generate new files accordingly.

First we need to import certain libraries:

```
# Import the following to classify the new seed words
from sklearn.naive_bayes import GaussianNB
import numpy as np
```

We then need to create a training matrix and training vector which holds the labels/scores for the matrix. We will be training our classifier on the seed words, since we know their labels, with the goal of predicting the new words in the future. Each seed word is a feature/dimension, therefore the matrix will have seedCount rows and seedCount columns. The row represents which seed word’s features we are seeing, and the column represents which seed word it is being related to. The numerical value in each cell tells us the type of relationships (and/but).

```
# Create a square matrix of size seedCt x seedCt to insert training samples
allSeeds = pos_words + neg_words
seedCt = len(allSeeds)
trainMatrix = np.zeros((seedCt, seedCt))
#Create vector of length seedCt to label training features
trainScores = np.zeros(seedCt)
```

We now need to populate the training matrix with the positive seed words. We don’t worry about populating `trainScores` since it already holds 0s for these indices which will be our label for positive sentiment.

```
# Populate train matrix with positive seed words
for rowA, word in enumerate(pos_words):
    if word in distance_matrix:
        columns = distance_matrix[word]
        for seedWord in columns.keys():
            columnNum = allSeeds.index(seedWord)
            trainMatrix[rowA][columnNum] = columns[seedWord]
```

We now do the same for the negative seed words, making sure to change the `trainScores` index to 1 as we do so.

```
#Populate train matrix with negative seed words
rowA += 1
```

```
for rowB, word in enumerate(neg_words):
    if word in distance_matrix:
        columns = distance_matrix[word]
        for seedWord in columns.keys():
            columnNum = allSeeds.index(seedWord)
            trainMatrix[rowA+rowB][columnNum] = columns[seedWord]
    trainScores[rowA+rowB] = 1
```

Before we train, we must ensure that each seed word also correlates to itself to some degree. The following gives each seed word a score of 1 with itself.

```
#Ensure every seed word correlates to itself
np.fill_diagonal(trainMatrix, 1)
```

Now we train our Naive Bayes Classifier.

```
#Train a naive bayes classifier on the seed words
clf = GaussianNB()
clf.fit(trainMatrix, trainScores)
```

Now, we will use this classifier to score each new word and determine if it is positive, negative, or neutral. We currently decide this rating by seeing where the scores lie among a certain threshold. Play around with this threshold value and see how the seed words change. A higher threshold should be more selective in which words it keeps.

```
#Parse through each new word
for word in distance_matrix.keys():
    if word not in allSeeds:
        row = np.zeros(seedCt)
        columns = distance_matrix[word]
        #Load in each seed word's features into a vector of length == seedCt
        for seedWord in columns.keys():
            columnNum = allSeeds.index(seedWord)
            row[columnNum] = columns[seedWord]
        row = [list(row)]
        #Determine if it is positive, negative or neutral based on its score
        scores = clf.predict_proba(row)[0]
        diff = scores[0] - scores[1]
        threshold = 0.1
        if diff > threshold:
            pos_words.append(word)
        elif diff < (threshold*(-1)):
            neg_words.append(word)
```

Finally we print out our new seeds.

```
#Print out the new and old seeds
```

```
with open("new_pos_words.txt",'w') as fout:
    for word in pos_words:
        fout.write(str(word) + "\n")
with open("new_neg_words.txt",'w') as fout:
    for word in neg_words:
        fout.write(str(word) + "\n")
print('done!')
```

Step 5: Rerun the rating

Lets see if our new seed words are helpful. Run the rateLexicon script again as follows:

```
> python3 rateLexicon.py new_pos_words.txt new_neg_words.txt
```

You should get an accuracy of about 85% which is a significant improvement. You might be thinking that this improvement is inevitable as we filter out less of each text for the classifier to train and test on. However, we are still filtering out a vast majority of the words in each text by highlighting only these annotated words as features, allowing for the classifier to run faster and focus on the most informative features. For this reason, an improvement of 5% implies we've efficiently added more features for the classifier to focus on.