Xuewen Sherry Li
CS165A

Architecture

Classes:

`class Stemmer -` contains methods to return the stem of a word
code source:
github.com/aterkik/stemmer/blob/f7759db67381468fe751aac6feeb46603b1ecbd5/stemmer.py.

Methods:

`def parse(filename, train_or_test=None) -` reads in training data or testing data files and returns reviews, represented as a list of lists in which each inner list contains cleaned words for each review. If train_or_test == 'train', then this function will group words into positive_words or negative_words based on the given labels. positive_words and negative_words are both dictionary types whose key is the word and value is the frequency(word).

`def clean(list_of_words) -` takes in a list of words and returns the list without stop words and maps commonly-stemmed words all to the same stem. Technically, this is creating a list of clean unigrams.

`def uni_and_bi_grams(list_of_words) -` takes in a list of words and returns the a list of unigrams and bigrams. First, it calls `clean()` to retrieve the list of unigrams, then it creates and appends bigrams to that same list thus producing a comprehensive unigram + bigram list.

`def remove_highly_correlated_features() -` removes words that appear around the same frequency in both classes

`def remove_rare_words() -` removes words with frequency equal to 1

`def make_prediction(list_of_words_by_review) -` takes in reviews (represented as a list of lists in which each inner list contains cleaned words for each review) and returns a list of predicted labels for each respective review. This is where I perform Naive Bayes classification using the word count data from in parse().

`def getAccuracy(file_with_true_labels, prediction_labels) -` takes in a file that contains true labels and the predicted labels list from make_prediction(), and returns the accuracy of prediction_labels. This function simply keeps count of how many entries from prediction_labels matches the list of true labels (parsed from file_with_true_labels) and divides it by the total number of labels.

Preprocessing

After parsing the file into individual reviews, I split each review into a list of words cleared of capitalizations and punctuations. I then call `uni_and_bi_grams(reviews)`. If it's a training set, I also remove highly correlated features.

For example, consider the following reviews:

HORRIBLE hate it 0

loving it! 1

`reviews` before `clean()` and `uni_bi_tri_grams()`:

`[['horrible', 'hate', 'it'], ['loving', 'it']]`

Reviews after:

`[['horribl', 'hate', ('horribl', 'hate')], ['love']]`

Model Building

After, `parse()` re-iterates through each review and appends words into positive_words or negative_words based on the review's label. positive_words and negative_words are both dictionary types in which the key is the word, and the value is the frequency(word). This grants us the word count of all the words in the vocabulary as well as word count for positive and negative words.This is necessary to compute the probabilities of new reviews being classified into each class later.

After parsing the training/testing data, I call `make_prediction(list_of_words_by_review)` which will take in a list of lists where each inner list contains words from each review, and return a list of predicted labels (1 for positive, 0 for negative). I find the maximum-a-posteriori class for each review x by finding: max[P(x | positive)P(positive), P(x | negative)P(negative)].
More specifically, the prior probability is:

$$\frac{\text{\# of words in positive\_words (or negative\_words)}}{\text{total \# of words in both positive\_words and negative\_words}}$$

and the posterior probability (with smoothing) for each word is:

$$\frac{\text{frequency(word) in positive\_words} + \alpha}{\text{\# of words in positive\_words} + \alpha|\text{vocabulary}|}$$

I chose $\alpha = 2.6$ because I found that any higher or lower values for α reduced the accuracy. Then, to avoid floating point underflow, I sum the logs of posterior probabilities for each word in the review to get the overall posterior probability of the review itself. Finally, I add the posterior probability of each review with the logged prior probability for each class (positive or negative) to determine the maximum-a-posterior class.

Results

20 seconds (training)

5 seconds (labeling)
0.9998 (training)
0.906 (testing)

Top 10 important features:

(frequency(feature) in positive reviews, feature)	(frequency(feature) in negative reviews, feature)
(1586, ('is', 'a')) (1496, ('and', 'the')) (1484, 'when') (1479, 'just') (1478, 'well') (1459, 'see') (1428, 'if') (1385, ('to', 'the')) (1358, 'great') (1350, 'stori')	(1996, 'just') (1852, 'if') (1718, 'bad') (1567, 'no') (1524, 'make') (1488, 'even') (1466, 'would') (1409, 'onli') (1334, ('to', 'be')) (1309, 'were')

Challenges

The most time-consuming part of my project was figuring out how to design my parse() function such that it handles training sets and testing sets separately. It was also difficult to write an effective word-stemming program myself because my program would map “loving” to “lov”, therefore I found a stronger program on Github that is able to recognize “loving” as “love.” It also took a dozen tries find the best smoothing values for my posterior probabilities. I learned that more stop words does not mean better. Lastly, I was stuck for a long time because my improvement plateaued however, learning to work with bigrams and removing highly correlated features helped.

Weaknesses

Although my stopwords list is long, it is not comprehensive. My program is also slower than I’d expected (expected ~3 seconds max but it took 11 seconds). I believe this is mainly due to reading in each line for a file, passing each line into several separate functions (clean(), word_stem()), and then finally appending each result into a separate list. This makes my program more memory-expensive and I am sure there are more optimal ways that can parse and clean in shorter run time while using less space. Additionally, the portion of my code that trains the data is inside of parse(), which semantically doesn’t make much sense. It should be in a separate “train” function, but due to the very specific way I wrote my code, I need to utilize the local variables from parse() thus my code loses semantic readability.