Samuel L. Peoples
IST 664: Natural Language Processing
Final Project
22 December 2018

Sentiment Classification of Movie Reviews

This dataset was produced for the Kaggle competition, described here: https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews, and which uses data from the sentiment analysis by Socher et al, detailed at this web site: http://nlp.stanford.edu/sentiment/. The data was taken from the original Pang and Lee movie review corpus based on reviews from the Rotten Tomatoes web site. Socher's group used crowd-sourcing to manually annotate all the subphrases of sentences with a sentiment label ranging over: "negative", "somewhat negative", "neutral", "somewhat positive", "positive".

Although the actual Kaggle competition is over, the data is still available at https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews/data. We are going to use the training data "train.tsv", and some test data is also available "test.tsv". There appear to be 156,060 phrases in the training data file, and one of the challenges will be to choose an appropriate subset for processing and training.

The experiments that will be tested will be on three different preprocessing steps on unigram (word frequency) features, including filtered based on longest word and phrases less than three words in length, no preprocessing, and the removal of stopwords. Once a well-performing baseline is selected, bigrams, POS tags, subjectivity sentiment (positive, negative, neutral), LIWC sentiment (positive, negative), sentiment (both subjectivity and LIWC), and all combined feature sets are to be tested using NLTK's Naïve Bayes classifier.

The first step was to load the data with a limit of 10,000. Throughout the training process, it was shown that filtering the data would improve the ultimate results. It was noticed that the SentenceIds had long sentences that were reduced in length iteratively, so the longest sentence, and any phrase less than or equal to three words in length were filtered.

```
# convert the limit argument from a string to an int
limit = int(limitStr)

os.chdir(dirPath)

df = pd.DataFrame.from_csv('./train.tsv', sep='\t')
phrasedata = []
for sid in df.SentenceId.unique():
    df_temp = df.loc[df['SentenceId'] == sid]
    phrases = []
    sentiment = []
    for p in df_temp['Phrase']:
        phrases.append(p)
    for s in df_temp['Sentiment']:
        sentiment.append(s)
    max_len = 0
    index = 0
    i = 0
    for phrase in phrases:
        if len(phrase) > max_len:
            max_len = len(phrase)
            index = i
        i += 1
    phrasedata.append((phrases[index], sentiment[index]))
    for i in range(len(phrases)):
        if len(phrases[i].split(" ")) <= 3:
            phrasedata.append((phrases[i], sentiment[i]))

f= open('./train.tsv', 'r')
```

```
# loop over lines in the file and use the first limit of them
phrasedata_full = []
for line in f:
  # ignore the first line starting with Phrase and read all lines
  if (not line.startswith('Phrase')):
    # remove final end of line character
    line = line.strip()
    # each line has 4 items separated by tabs
    # ignore the phrase and sentence ids, and keep the phrase and sentiment
    phrasedata_full.append(line.split('\t')[2:4])

# pick a random sample of length limit because of phrase overlapping sequences
random.shuffle(phrasedata)
random.shuffle(phrasedata_full)

phraselist = phrasedata[:limit]
phraselist_2 = phrasedata_full[:limit]
Read 72040 filtered phrases, using 10000 random phrases
Read 156060 phrases, using 10000 random phrases
```

It was then necessary to tokenize and further filter the data. Multiple vocabularies are developed to support ‘non-preprocessed’ and ‘preprocessed’ data for comparison between various feature sets. Punctuation is removed, as well as stopwords, and non-english words (typos). The three documents, representing ‘non-preprocessed filtered data’, ‘preprocessed filtered data’, and ‘random preprocessed data’ are finally prepared. Next the frequency distribution, and bigram features are generated.

```
# punctuation
temp_docs = []
for item in phrasedocs:
    temp = []
    for token in item[0]:
        matchObj = re.search('[^a-zA-Z\d]', token)
        if not matchObj:
            temp.append(token.lower())
    temp_docs.append((temp, item[1]))
phrasedocs = temp_docs
temp_docs = []
for item in phrasedocs_2:
    temp = []
    for token in item[0]:
        matchObj = re.search('[^a-zA-Z\d]', token)
        if not matchObj:
            temp.append(token.lower())
    temp_docs.append((temp, item[1]))
phrasedocs_2 = temp_docs
# stopwords, typos
sw = set(stopwords.words('english'))
phrasedocs_pp = []
all_words_list = []
phrasedocs_2_pp = []
all_words_list_2 = []
for item in phrasedocs:
  temp = []
  for token in item[0]:
    all_words_list.append(token.lower())
    if token not in sw and wordnet.synsets(token):
      temp.append(token)
    phrasedocs_pp.append((temp,item[1]))
for item in phrasedocs_2:
  temp = []
  for token in item[0]:
    all_words_list_2.append(token.lower())
    if token not in sw and wordnet.synsets(token):
```

```
      temp.append(token)
    phrasedocs_2_pp.append((temp,item[1]))
documents = phrasedocs
documents_3 = phrasedocs_pp
documents_3f = phrasedocs_2
# continue as usual to get all words and create word features
all_words = nltk.FreqDist(all_words_list)
all_words_2 = nltk.FreqDist(all_words_list_2)
# get the 1500 most frequently appearing keywords in the corpus
word_items = all_words.most_common(1500)
word_features = [word for (word, count) in word_items]
word_items_2 = all_words_2.most_common(1500)
# adding bigram features
bigram_measures = nltk.collocations.BigramAssocMeasures()
# create the bigram finder on all the words in sequence
finder = BigramCollocationFinder.from_words(all_words_list)
finder_2 = BigramCollocationFinder.from_words(all_words_list_2)
# define the top 500 bigrams using the chi squared measure
bigram_features = finder.nbest(bigram_measures.chi_sq, 500)
```

The various featuresets are developed, displaying the accuracy of Unigrams with no preprocessing, filtered and from the full data set, and with preprocessing (filtered). Using two-fold cross validation and NLTK's Naïve Bayes classifier, the filtered data performed the best, without preprocessing with a mean accuracy of 62.54%.

```
featuresets_2 = [(document_features(d, word_features), c) for (d, c) in documents_3f]
featuresets = [(document_features(d, word_features), c) for (d, c) in documents]
featuresets_3 = [(document_features(d, word_features), c) for (d, c) in documents_3]
cross_validation_accuracy(2, featuresets_2)
cross_validation_accuracy(2, featuresets)
cross_validation_accuracy(2, featuresets_3)
```

```
Generating Feature sets for:
       Unigrams, no preprocessing, from full dataset
       Unigrams, no preprocessing, filtered
       Unigrams, with preprocessing, filtered
word frequencies, from full dataset
Each fold size: 5000
    0 0.5178
               Precision      Recall         F1
       0             0.154      0.161      0.157
       1             0.234      0.355      0.282
       2             0.830      0.601      0.697
       3             0.222      0.405      0.286
       4             0.194      0.326      0.244
    1 0.5276
               Precision      Recall         F1
       0             0.184      0.192      0.188
       1             0.239      0.358      0.287
       2             0.816      0.622      0.706
       3             0.238      0.376      0.291
       4             0.208      0.313      0.250
    mean accuracy 0.5226999999999999
```

```
word frequencies, filtered
Each fold size: 5000
    0 0.6264
               Precision      Recall         F1
        0            0.222      0.187      0.203
        1            0.094      0.364      0.149
        2            0.938      0.688      0.794
        3            0.155      0.446      0.230
        4            0.159      0.211      0.181
    1 0.6244
               Precision      Recall         F1
        0            0.188      0.129      0.153
        1            0.077      0.405      0.129
        2            0.940      0.685      0.792
        3            0.125      0.396      0.190
        4            0.128      0.211      0.160
    mean accuracy 0.6254
word frequencies, with preprocessing, filtered
Each fold size: 18004
    0 0.4379582315041102
               Precision      Recall         F1
        0            0.104      0.163      0.127
        1            0.267      0.283      0.275
        2            0.757      0.560      0.644
        3            0.280      0.355      0.313
        4            0.170      0.305      0.218
    1 0.4401244167962675
        Precision      Recall         F1
        0            0.176      0.246      0.205
        1            0.188      0.308      0.234
        2            0.746      0.557      0.638
        3            0.282      0.311      0.296
        4            0.188      0.252      0.215
    mean accuracy 0.4390413241501888
```

Models were then developed for feature sets based on bigrams, part of speech tagging, subjectivity, LIWC, both subjectivity, and all features combined. The highest performing model was based on the sentiment features, which included both the LIWC and subjectivity data. The precision, recall, and F1 values clustered around the neutral responses, which was expected because the data has a normal distribution about the neutral responses.

```
bigram_featuresets = [(bigram_document_features(d,
                       word_features, bigram_features), c) for (d, c) in documents]
POS_featuresets = [(POS_features(d, word_features), c) for (d, c) in documents]
subj_featuresets = [(subj_features(d, word_features), c) for (d, c) in documents]
LIWC_featuresets = [(LIWC_features(d, word_features), c) for (d, c) in documents]
sent_featuresets = [(sent_features(d, word_features), c) for (d, c) in documents]
all_featuresets = [(all_features(d, word_features, bigram_features), c) for (d, c) in documents]
cross_validation_accuracy(2, bigram_featuresets)
cross_validation_accuracy(2, POS_featuresets)
cross_validation_accuracy(2, subj_featuresets)
cross_validation_accuracy(2, LIWC_featuresets)
cross_validation_accuracy(2, sent_featuresets)
cross_validation_accuracy(2, all_featuresets)
Generating Feature sets for:
        Bigrams, no preprocessing, filtered
        POS Tags, no preprocessing, filtered
        Subjectivity, no preprocessing, filtered
        LIWC, no preprocessing, filtered
        Sentiment, no preprocessing, filtered
        All Features, with preprocessing, filtered
```

```
bigram, without preprocessing, filtered
Each fold size: 5000
    0 0.6274
               Precision      Recall         F1
        0           0.225      0.183      0.202
        1           0.086      0.306      0.134
        2           0.939      0.683      0.791
        3           0.138      0.446      0.210
        4           0.147      0.269      0.190
    1 0.6248
               Precision      Recall         F1
        0           0.230      0.152      0.183
        1           0.091      0.396      0.149
        2           0.937      0.690      0.795
        3           0.133      0.406      0.200
        4           0.174      0.221      0.195
    mean accuracy 0.6261

pos, without preprocessing, filtered
Each fold size: 5000
    0 0.6238
               Precision      Recall         F1
        0           0.245      0.167      0.199
        1           0.092      0.297      0.141
        2           0.935      0.687      0.792
        3           0.122      0.437      0.191
        4           0.157      0.246      0.192
        1 0.6202
               Precision      Recall         F1
        0           0.257      0.137      0.178
        1           0.082      0.341      0.132
        2           0.931      0.692      0.794
        3           0.128      0.399      0.194
        4           0.188      0.258      0.218
    mean accuracy 0.622

subjectivity, without preprocessing, filtered
Each fold size: 5000
    0 0.6412
               Precision      Recall         F1
        0           0.225      0.194      0.209
        1           0.124      0.372      0.186
        2           0.923      0.705      0.799
        3           0.228      0.488      0.310
        4           0.198      0.283      0.233
    1 0.6384
               Precision      Recall         F1
        0           0.257      0.165      0.201
        1           0.122      0.431      0.191
        2           0.910      0.714      0.800
        3           0.268      0.485      0.345
        4           0.213      0.246      0.228
    mean accuracy 0.6397999999999999
```

```
LIWC, without preprocessing, filtered
Each fold size: 5000
    0 0.6282
               Precision      Recall         F1
       0             0.232      0.194      0.211
       1             0.086      0.313      0.135
       2             0.936      0.686      0.791
       3             0.150      0.440      0.224
       4             0.157      0.258      0.195
    1 0.6288
               Precision      Recall         F1
       0             0.230      0.148      0.180
       1             0.105      0.424      0.169
       2             0.932      0.697      0.797
       3             0.161      0.426      0.234
       4             0.174      0.224      0.196
    mean accuracy 0.6285000000000001

sentiment, without preprocessing, filtered
Each fold size: 5000
    0 0.6408
               Precision      Recall         F1
       0             0.225      0.191      0.207
       1             0.123      0.369      0.184
       2             0.923      0.705      0.799
       3             0.230      0.491      0.313
       4             0.194      0.278      0.228
    1 0.6386
               Precision      Recall         F1
       0             0.257      0.162      0.199
       1             0.121      0.426      0.188
       2             0.911      0.715      0.801
       3             0.269      0.484      0.346
       4             0.208      0.243      0.224
    mean accuracy 0.6396999999999999

all features, without preprocessing, filtered
Each fold size: 5000
0 0.63
        Precision      Recall         F1
0             0.258      0.176      0.209
1             0.138      0.341      0.197
2             0.906      0.707      0.794
3             0.210      0.458      0.288
4             0.198      0.276      0.231
1 0.625
        Precision      Recall         F1
0             0.270      0.149      0.192
1             0.121      0.336      0.178
2             0.892      0.716      0.794
3             0.247      0.451      0.319
4             0.232      0.274      0.251
mean accuracy 0.6275
```

The SK Learn Naïve Bayes classifier was attempted, and blocks were raised in formatting the data, and the full dataset was chosen for implementation. The preprocessing steps were carried over from the highest performing parameters previously, being the removal of punctuation, and the tokenization of all phrases. Sentiment features are used because they performed with 63.96% accuracy on the 10,000-phrase sample, the highest performing model and feature set. The output was saved to a csv file, and merged with the sample submission file to be verified on Kaggle. Unfortunately, the final length of the prediction was less than the length of the expected by roughly three thousand entries, which prevented the accuracy from being higher than 49%.

Name	Submitted	Wait time	Execution time	Score
sampleSubmission.csv	just now	1 seconds	1 seconds	0.49002

```
df_train = pd.DataFrame.from_csv('./train.tsv', sep='\t')
df_test = pd.DataFrame.from_csv('./test.tsv', sep='\t')
phrases_train = list(df_train['Phrase'])
sentiment_train = list(df_train['Sentiment'])
phrases_test = list(df_test['Phrase'])
phrasedata_train = []
for i in range(len(phrases_train)):
    phrasedata_train.append((phrases_train[i],sentiment_train[i]))
phrasedata_test = []
for i in range(len(phrases_test)):
    phrasedata_test.append(phrases_test[i])
phraselist_train = phrasedata_train
phraselist_test = phrasedata_test
phrasedocs_train = []
for phrase in phraselist_train:
  tokens = nltk.word_tokenize(phrase[0])
  phrasedocs_train.append((tokens, int(phrase[1])))
phrasedocs_test = []
for phrase in phraselist_test:
  tokens = nltk.word_tokenize(phrase[0])
  phrasedocs_test.append((tokens))
temp_docs = []
for item in phrasedocs_train:
    temp = []
    for token in item[0]:
        matchObj = re.search('[^a-zA-Z\d]', token)
        if not matchObj:
            temp.append(token.lower())
    temp_docs.append((temp, item[1]))
phrasedocs_train = temp_docs
temp_docs = []
for item in phrasedocs_test:
    temp = []
    for token in item:
        matchObj = re.search('[^a-zA-Z\d]', token)
        if not matchObj:
            temp.append(token.lower())
    temp_docs.append(temp)
phrasedocs_test = temp_docs
sw = set(stopwords.words('english'))
phrasedocs_train_2 = []
all_words_list_train = []
for item in phrasedocs_train:
  temp = []
  for token in item[0]:
    all_words_list_train.append(token.lower())
    if wordnet.synsets(token):
      temp.append(token)
    phrasedocs_train_2.append((temp,item[1]))
phrasedocs_test_2 = []
all_words_list_test = []
for item in phrasedocs_test:
  temp = []
  for token in item:
      all_words_list_test.append(token.lower())
      if wordnet.synsets(token):
          temp.append(token)
      phrasedocs_test_2.append(temp)
documents_train = phrasedocs_train_2
documents_test = phrasedocs_test_2
all_words_train = nltk.FreqDist(all_words_list_train)
all_words_test = nltk.FreqDist(all_words_list_test)
```

```
word_items_train = all_words_train.most_common(1500)
word_features_train = [word for (word, count) in word_items_train]
word_items_test = all_words_test.most_common(1500)
word_features_test = [word for (word, count) in word_items_test]
bigram_measures = nltk.collocations.BigramAssocMeasures()
finder_train = BigramCollocationFinder.from_words(all_words_list_train)
finder_test = BigramCollocationFinder.from_words(all_words_list_test)
bigram_features_train = finder_train.nbest(bigram_measures.chi_sq, 500)
bigram_features_test = finder_test.nbest(bigram_measures.chi_sq, 500)
documents_train = documents_train[0:10000]
all_featuresets_train = [(sent_features(d, word_features_train), c) for (d, c) in
documents_train]
all_featuresets_test = [(sent_features(d, word_features_test)) for (d) in documents_test]
accuracy_full(all_featuresets_train, all_featuresets_test)
```

```
Generating Feature sets for:
        Sentiment, full sample training set

sentiment, with preprocessing, Full dataset
```

The feature set function definitions, evaluation, and cross-validation function definitions are included below:

```
import sentiment_read_subjectivity
(positivelist, neutrallist, negativelist) =
    sentiment_read_subjectivity.read_subjectivity_three_types(
      'SentimentLexicons/subjclueslen1-HLTEMNLP05.tff')
import sentiment_read_LIWC_pos_neg_words
(poslist, neglist) = sentiment_read_LIWC_pos_neg_words.read_words()

def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    return features

def bigram_document_features(document, word_features, bigram_features):
    document_words = set(document)
    document_bigrams = nltk.bigrams(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    for bigram in bigram_features:
        features['B_{}_{}'.format(bigram[0], bigram[1])] = (bigram in document_bigrams)
    return features

def POS_features(document, word_features):
    document_words = set(document)
    tagged_words = nltk.pos_tag(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    numNoun = 0
    numVerb = 0
    numAdj = 0
    numAdverb = 0
    for (word, tag) in tagged_words:
        if tag.startswith('N'): numNoun += 1
        if tag.startswith('V'): numVerb += 1
        if tag.startswith('J'): numAdj += 1
        if tag.startswith('R'): numAdverb += 1
    features['nouns'] = numNoun
    features['verbs'] = numVerb
    features['adjectives'] = numAdj
    features['adverbs'] = numAdverb
    return features
```

```
def subj_features(document, word_features):
  document_words = set(document)
  features = {}
  for word in word_features:
    features['contains({})'.format(word)] = (word in document_words)
  numPos = 0
  numNeg = 0
  numNeu = 0
  for word in document_words:
    if word in positivelist: numPos += 1
    if word in negativelist: numNeg += 1
    if word in neutrallist: numNeu += 1
  features['positive'] = numPos
  features['negative'] = numNeg
  features['neutral'] = numNeu
  return features

def LIWC_features(document, word_features):
  document_words = set(document)
  features = {}
  for word in word_features:
    features['contains({})'.format(word)] = (word in document_words)
  numPos = 0
  numNeg = 0
  for word in document_words:
    if word in poslist: numPos += 1
    if word in neglist: numNeg += 1
  features['positive'] = numPos
  features['negative'] = numNeg
  return features

def sent_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    numPos = 0
    numNeg = 0
    numNeu = 0
    for word in document_words:
        if word in poslist or word in positivelist: numPos += 1
        if word in neglist or word in negativelist: numNeg += 1
        if word in neutrallist: numNeu += 1

    features['positive'] = numPos
    features['negative'] = numNeg
    features['neutral'] = numNeu
    return features

def all_features(document, word_features, bigram_features):
  document_words = set(document)
  features = {}
  tagged_words = nltk.pos_tag(document)
  document_bigrams = nltk.bigrams(document)
  for word in word_features:
    features['contains({})'.format(word)] = (word in document_words)
  numPos = 0
  numNeg = 0
  numNeu = 0
  numNoun = 0
  numVerb = 0
  numAdj = 0
  numAdverb = 0

  for word in document_words:
    if word in poslist or word in positivelist: numPos += 1
    if word in neglist or word in negativelist: numNeg += 1
    if word in neutrallist: numNeu += 1
```

```
  for (word, tag) in tagged_words:
      if tag.startswith('N'): numNoun += 1
      if tag.startswith('V'): numVerb += 1
      if tag.startswith('J'): numAdj += 1
      if tag.startswith('R'): numAdverb += 1

  for word in word_features:
      features['V_{}'.format(word)] = (word in document_words)
  for bigram in bigram_features:
      features['B_{}_{}'.format(bigram[0], bigram[1])] = (bigram in document_bigrams)
  features['nouns'] = numNoun
  features['verbs'] = numVerb
  features['adjectives'] = numAdj
  features['adverbs'] = numAdverb
  features['positive'] = numPos
  features['negative'] = numNeg
  features['neutral'] = numNeu
  return features

def eval_measures(gold, predicted):
    labels = list(set(gold))
    recall_list = []
    precision_list = []
    F1_list = []
    for lab in labels:
        TP = FP = FN = TN = 0
        for i, val in enumerate(gold):
            if val == lab and predicted[i] == lab:  TP += 1
            if val == lab and predicted[i] != lab:  FN += 1
            if val != lab and predicted[i] == lab:  FP += 1
            if val != lab and predicted[i] != lab:  TN += 1
        if (TP + FP) == 0:
            recall = 0
        else:
            recall = TP / (TP + FP)
        if (TP + FN) == 0:
            precision = 0
        else:
            precision = TP / (TP + FN)
        recall_list.append(recall)
        precision_list.append(precision)
        if (recall + precision) == 0:
            F1_list.append(0)
        else:
            F1_list.append( 2 * (recall * precision) / (recall + precision))
    print('\tPrecision\tRecall\t\tF1')
    for i, lab in enumerate(labels):
        print(lab, '\t', "{:10.3f}".format(precision_list[i]), \
          "{:10.3f}".format(recall_list[i]), "{:10.3f}".format(F1_list[i]))

def cross_validation_accuracy(num_folds, featuresets):
    subset_size = int(len(featuresets)/num_folds)
    print('Each fold size:', subset_size)
    accuracy_list = []
    for i in range(num_folds):
        test_this_round = featuresets[(i*subset_size):][:subset_size]
        train_this_round = featuresets[:(i*subset_size)] + featuresets[((i+1)*subset_size):]
        classifier = nltk.NaiveBayesClassifier.train(train_this_round)
        accuracy_this_round = nltk.classify.accuracy(classifier, test_this_round)
        print (i, accuracy_this_round)
        accuracy_list.append(accuracy_this_round)
        goldlist = []
        predictedlist = []
        for (features, label) in test_this_round:
            goldlist.append(label)
            predictedlist.append(classifier.classify(features))
        eval_measures(goldlist, predictedlist)
    print ('mean accuracy', sum(accuracy_list) / num_folds)
```

```
def accuracy_full(featuresets_train, featuresets_test):
    classifier = nltk.NaiveBayesClassifier.train(featuresets_train)
    prediction = []
    for feature in featuresets_test:
        prediction.append(classifier.classify(feature))
    with open('./prediction.csv','w') as f:
        for item in prediction:
            f.write(str(item)+"\n")
```

Ultimately, a feature set containing the associations with the subjectivity positive, negative and neutral lists, and the LIWC positive and negative lists allowed for a sample accuracy performance of 63.96%, and an overall accuracy of 49%. With more tuning, the final accuracy could be improved, and brought to match or exceed the sample performance. When comparing this theoretical performance to the top performing kernels on Kaggle, it is within 10% accuracy, which provides some relief that these methods are approaching the ‘right track’. The removal of stopwords making the models perform worse was interesting, and I was eager to see results without them. I was hopeful the combined feature sets would perform with higher accuracy, but would probably revisit the sampling to increase the accuracy overall. When working with the feature sets, a uniform distribution was considered, but models performed with only 40% accuracy, which was interesting. The precision and recall clustering near the largest sampled sentiments is expected and is contributing to some overfitting.