NLP Lab Session
Week 11, November 9, 2017
Bigram and POS tag feature sets and cross-validation in the NLTK

Getting Started

For this lab session download the examples: LabWeek11bigrams.ipynb or .py and put it in your class folder for copy/pasting examples. Start your Python interpreter session.

```
>>> import nltk
```

In this week's lab, we show two more types of features sometimes used in classification and how to use more classifier evaluation measures. After this week's lab, you should be able to use a variety of features to test with your final project data, and also be able to report better evaluation measures.

Bigram Features

One more important source of features often used in sentiment and other document or sentence-level classifications is bigram features. Typically these features are added to word level features.

First, we restart by loading the movie review sentences and getting the baseline performance of the unigram features. This is a repeat from last week in order to get started, except that we will change the size of the feature sets, and the training and test sets.

```
>>> from nltk.corpus import sentence_polarity
>>> import random
```

The movie review documents are not labeled individually, but are separated into file directories by category. We first create the list of documents/sentences where each is paired with its label.

```
>>> documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]
```

In this list, each item is a pair (d,c) where d is a list of words from a sentence and c is its label, either 'pos' or 'neg'.

Since the documents are in order by label, we mix them up for later separation into training and test sets.

```
>>> random.shuffle(documents)
```

We need to define the set of words that will be used for features. For this week's lab, we will limit the length of the word features to 1500.

```
>>> all_words_list = [word for (sent,cat) in documents for word in sent]
```

```
>>> all_words = nltk.FreqDist(all_words_list)
>>> word_items = all_words.most_common(1500)
>>> word_features = [word for (word, freq) in word_items]
```

As before, the word feature labels will be 'contains(keyword)' for each keyword (aka word) in the word_features set, and the value of the feature will be Boolean, according to whether the word is contained in that document.

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

Define the feature sets for the documents.

```
>>> featuresets = [(document_features(d), c) for (d,c) in documents]
>>> len(featuresets)
```

We create the training and test sets, train a Naïve Bayes classifier, and look at the accuracy. This time, we separate the data into a 90%, 10% split for training and testing, which is more usual.

```
>>> train_set, test_set = featuresets[1000:], featuresets[:1000]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
```

Now that we have a baseline for performance for this random split of the data, we'll create some bigram features.

As we saw in the lab in Week 3, when we worked on generating bigrams from documents, if we want to use highly frequent bigrams, we need to filter out special characters, which were very frequent in the bigrams, and also filter by frequency. The bigram pmi measure also required some filtering to get frequent and meaningful bigrams. But there is another bigram association measure that is more often used to filter bigrams for classification features. This is the chi-squared measure, which is another measure of information gain, but which does its own frequency filtering. Another frequently used alternative is to just use frequency, which is the bigram measure raw_freq.

We'll start by importing the collocations package and creating a short cut variable name for the bigram association measures.

```
>>> from nltk.collocations import *
>>> bigram_measures = nltk.collocations.BigramAssocMeasures()
```

We create a bigram collocation finder using the original movie review words, since the bigram finder must have the words in order. Note that our all_words_list has exactly this list.

```
>>> all_words_list[:50]
>>> finder = BigramCollocationFinder.from_words(all_words_list)
```

We use the chi-squared measure to get bigrams that are informative features. Note that we don't need to get the scores of the bigrams, so we use the nbest function which just returns the highest scoring bigrams, using the number specified.
(Or try bigram_measures.raw_freq.)

```
>>> bigram_features = finder.nbest(bigram_measures.chi_sq, 500)
```

The nbest function returns a list of significant bigrams in this corpus, and we can look at some of them.

```
>>> bigram_features[:50]
```

Now we create a feature extraction function that has all the word features as before, but also has bigram 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['contains({})'.format(word)] = (word in document_words)
        for bigram in bigram_features:
                features['bigram({} {})'.format(bigram[0], bigram[1])] = (bigram in
document_bigrams)
        return features
```

In this function, in order to test if any bigram in the bigram_features list is in the document, we need to generate the bigrams of the document, which we do using the nltk.bigrams function. To show this, we define a sentence and show the bigrams.

```
>>> sent = ['Arthur','carefully','rode','the','brown','horse','around','the','castle']
>>> sentbigrams = list(nltk.bigrams(sent))
>>> sentbigrams
[('Arthur', 'carefully'), ('carefully', 'rode'), ('rode', 'the'), ('the', 'brown'), ('brown',
'horse'), ('horse', 'around'), ('around', 'the'), ('the', 'castle')]
```

For any one bigram, we can test if it is in the bigrams of the sentence and we can use string formatting, with two occurrences of %s, to insert the two words of the bigram into the name of the feature.

```
>>> bigram = ('brown','horse')
>>> bigram in sentbigrams
True
>>> 'bigram({} {})'.format(bigram[0], bigram[1])
'bigram(brown horse)'
```

Now we create feature sets as before, but using this feature extraction function.

```
>>> bigram_featuresets = [(bigram_document_features(d), c) for (d,c) in documents]

There should be 2000 features:  1500 word features and 500 bigram features
>>> len(bigram_featuresets[0][0].keys())

>>> train_set, test_set = bigram_featuresets[200:], bigram_featuresets[:200]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
```

So in my random training, test split, the bigrams did not improve the classification.

POS tag features

There are some classification tasks where part-of-speech tag features can have an effect. In my experience, this is more likely for shorter units of classification, such as sentence level classification or shorter social media such as tweets.

The most common way to use POS tagging information is to include counts of various types of word tags. Here is an example feature function that counts nouns, verbs, adjectives and adverbs for features. [Note that this function calls nltk.pos_tag every time that it is run and for repeated experiments, you could pre-compute the pos tags and save them for every document.]

```
>>> def POS_features(document):
        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

Try out the POS features.
>>> POS_featuresets = [(POS_features(d, word_features), c) for (d, c) in documents]
# number of features for document 0
>>> len(POS_featuresets[0][0].keys())
```

```
# the first sentence
>>> documents[0]
(['simplistic', ',', 'silly', 'and', 'tedious', '.'], 'neg')
# the pos tag features for this sentence
>>> POS_featuresets[0][0]['nouns']
0
>>> POS_featuresets[0][0]['verbs']
0
>>> POS_featuresets[0][0]['adjectives']
2
>>> POS_featuresets[0][0]['adverbs']
1

>>> train_set, test_set = POS_featuresets[1000:], POS_featuresets[:1000]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
```

This improved classification a small amount for my train/test split.

Using Other Classifiers (for future reference)

We have been using the NaïveBayesClassifier for all our classification tasks so far. There are other classifiers that we can use in the NLTK classify package: http://www.nltk.org/api/nltk.classify.html. We show here how to call two other classifiers:
Decision Tree Classifier and Maximum Entropy (MaxEnt) Classifier. We won't actually run these in lab because the performance is so slow.

The Decision Tree Classifier (see Chapter 6 Section 4 in the NLTK Book) decides which label to assign on the basis of a tree structure, where branches correspond to conditions on feature values, and leaves correspond to label assignments.

```
>>>  classifier = nltk.DecisionTreeClassifier.train(train_set, entropy_cutoff =0,
support_cutoff =0)
>>> nltk.classify.accuracy(classifier, test_set)
```

The MaxEnt Classifier (see Chapter 6 Section 6 in the NLTK Book) uses search techniques to find a set of parameters that will maximize the performance of the classifier.

```
>>> from nltk.classify import MaxentClassifier
>>> classifier = nltk.MaxentClassifier.train(train_set, max_iter = 1)
```

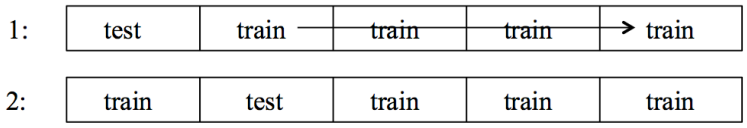
In the output, you will see the accuracy of the classifier on the train_set. If you do not provide a value for the max_iter parameter, the default max_iter is set to 100. You will most likely obtain a RunTime warning if you use the default max_iter so it is always better to set a smaller value for max_iter. You can then evaluate the accuracy of the MaxEnt Classifier on the test_set.

```
>>> nltk.classify.accuracy(classifier, test_set)
```

Cross-validation

As a final topic in evaluation, we have discussed that our testing of the features on the movie reviews and movie review sentences data is often skewed by the random sample. The remedy for this is to use different chunks of the data as the test set to repeatedly train a model and then average our performance over those models.

This method is called *cross-validation*, or sometimes *k-fold cross-validation*. In this method, we choose a number of folds, k, which is usually a small number like 5 or 10. We first randomly partition the development data into k subsets, each approximately equal in size. Then we train the classifier k times, where at each iteration, we use each subset in turn as the test set and the others as a training set.



. . .

NLTK does not have a built-in function for cross-validation, but we can program the process in a function that takes the number of folds and the feature sets, and iterates over training and testing a classifier.

```
>>> def cross_validation(num_folds, featuresets):
        subset_size = len(featuresets)/num_folds
        accuracy_list = []
        # iterate over the folds
        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:]
            # train using train_this_round
             classifier = nltk.NaiveBayesClassifier.train(train_this_round)
              # evaluate against test_this_round and save accuracy
              accuracy_this_round = nltk.classify.accuracy(classifier, test_this_round)
             print i, accuracy_this_round
            accuracy_list.append(accuracy_this_round)
        # find mean accuracy over all rounds
        print 'mean accuracy', sum(accuracy_list) / num_folds
```

Run the cross-validation on our word feature sets with 10 folds.

```
>>> cross_validation(10, featuresets)
```

Instead of accuracy, we should have a cross-validation function to report precision and recall for each label.

Exercise

In lab, we ran the cross_validation function on the variable feature_sets, which gave the average accuracy for the word feature sets. For your exercise, run the cross-validation function for the bigram feature sets and for the POS feature sets.

If you have time, run at least one of them for 10 folds instead of 5.

For each type of feature function definition, [ost into the discussion forum the original accuracy that we got with just one train/test split and compare it with cross-validation accuracy that you get for 5 folds. If you have time to run 10 fold cross-validation(s), post that as well.