NLP Lab Session
Week 12, November 18, 2015
Bigram Feature Sets and more evaluation in the NLTK

Getting Started

For this lab session download the examples: LabWeek12evaluate.txt and put it in your class folder for copy/pasting examples. Start your Python interpreter or IDLE 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 reviews 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 movie_reviews
>>> 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 where each document is paired with its label. There are 1000 documents labeled 'neg' and 1000 labeled 'pos'.

```
>>> documents = [(list(movie_reviews.words(fileid)), category)
            for category in movie_reviews.categories()
            for fileid in movie_reviews.fileids(category)]
```

In this list, each item is a pair (d,c) where d is a list of words from a movie review 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 = nltk.FreqDist(w.lower() for w in movie_reviews.words())
>>> word_features = all_words.keys()[:1500]
```

[As usual, for NLTK 3.x, use the most_common function to return the most frequent (word, frequency) pairs.]

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):
        document_words = set(document)
        features = {}
        for word in word_features:
                features['contains(%s)' % word] = (word in document_words)
        return features
```

Define the feature sets for the documents.

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

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[200:], featuresets[:200]
>>> 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.

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.

```
>>> finder = BigramCollocationFinder.from_words(movie_reviews.words())
```

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.

```
>>> 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):
        document_words = set(document)
        document_bigrams = nltk.bigrams(document)
        features = {}
        for word in word_features:
            features['contains(%s)' % word] = (word in document_words)
        for bigram in bigram_features:
            features['bigram(%s %s)' % bigram] = (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 = 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(%s %s)' % bigram
'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.

Now in the NLTK, this is difficult to demonstrate, since on my computer, it takes the default NLTK POS tagger about 30 minutes to tag the entire movie review corpus. If we were really going to use this, we should tag the corpus once and save it for later use with our testing.

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.

```
>>> def POS_features(document):
        document_words = set(document)
        tagged_words = nltk.pos_tag(document)
        features = {}
        for word in word_features:
            features['contains(%s)' % 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
```

Other Evaluation Measures

So far, we have been using simple accuracy for a performance evaluation measure of the predictive capability of the model that was learned from the training data. But we can learn more by looking at the predictions for each of the labels in our classifier.

We start by looking at the confusion matrix, which shows the results of a test for how many of the actual class labels (the gold standard labels) match with the predicted labels. In this diagram the two labels are called "Yes" and "No".

		Predicted Class	
		Class=Yes	Class=No
Actual Class	Class=Yes	a	b
	Class=No	c	d

a: TP (true positive)

b: FN (false negative)

c: FP (false positive)

d: TN (true negative)

When the predicted class is the same as the actual class, we call those examples the true positives. When the actual class was supposed to be Yes, but the predicted class was No, we call those examples the false negatives. When the actual class is No, but the classifier incorrectly predicted Yes, we call those examples the false positives. The true negatives are the remaining examples that were correctly predicted No. The number of each of these types of examples in the test set is put into the confusion matrix.

Note that the intuition for the terminology comes from the idea that we are trying to find all the examples where the class label is Yes, the positive examples. The false positives represent the positives which were predicted Wrong, and the false negatives represent the positives that were Missed. This idea originated in the Information Retrieval field where the Yes answers represented documents that were correctly retrieved as the result of a search.

In keeping with this intuition, two commonly used measures come from IR, where IR is only interested in the positive labels.

recall = TP / (TP + FP)   (the percentage of actual yes answers that are right)
precision = TP / (TP + FN) (the percentage of predicted yes answers that are right)

These two measures are sometimes combined into a kind of average, the harmonic mean, called the F-measure, which in its simplest form is:

F-measure = 2 * (recall * precision) / (recall + precision)

In situations where we are equally interested in correctly predicting Yes and No, and the numbers of these are roughly equal, then we may compute precision and recall for both the positive and negative labels. And we can also use the accuracy measure.

accuracy = TP + TN / (TP + FP + FN + TN)   (percentage of correct Yes and No out of all text examples)

In the NLTK, the confusion matrix is given by a function that takes two lists of labels for the test set. NLTK calls the first list the reference list, which is all the correct/gold labels for the test set, and the second list is the test list, which is all the predicted labels in the test set. These two lists are both in the order of the test set, so they can be compared to see which examples the classifier model agreed on or not.

First we build the reference and test lists from the classifier on the test set:

```
>>> reflist = []
>>> testlist = []
>>> for (features, label) in test_set:
        reflist.append(label)
        testlist.append(classifier.classify(features))
```

We can look at some examples.

```
>>> reflist[:30]
>>> testlist[:30]
```

Now we the NLTK function to define the confusion matrix, and we print it out:

```
>>> cm = nltk.metrics.ConfusionMatrix(reflist, testlist)
>>> print cm
    | n  p |
    | e  o |
    | g  s |
----+-------+
neg |<83>15 |
pos | 22<80>|
----+-------+
(row = reference; col = test)
```

Now our “pos” tags are in the second row and second column, so we may want to flip the terminology and say that there are 80 True Positives, 83 True Negatives, 15 False Positives and 22 False Negatives.

We could now just use some arithmetic to compute the precision and recall, but the NLTK has functions to do this. The per-label precision and recall functions expect to get a set of item identifiers that were gold labels and a set of item identifiers that were predicted labels.

```
>>> refpos = set()
>>> refneg = set()
>>> testpos = set()
>>> testneg = set()
>>> for i, label in enumerate(reflist):
        if label == 'neg': refneg.add(i)
        if label == 'pos': refpos.add(i)
>>> for i, label in enumerate(testlist):
        if label == 'neg': testneg.add(i)
        if label == 'pos': testpos.add(i)

>>> refpos
>>> testpos
```

Now to get precision, recall and F-measure for one of the labels, we must give the reference and test sets for that label. It is easiest to define a function that calls the three NLTK functions.

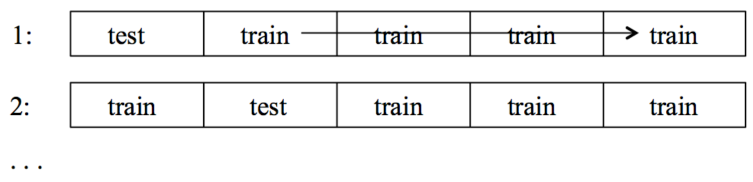
```
>>> def printmeasures(label, refset, testset):
        print label, 'precision:', nltk.metrics.precision(refset, testset)
        print label, 'recall:', nltk.metrics.recall(refset, testset)
        print label, 'F-measure:', nltk.metrics.f_measure(refset, testset)

>>> printmeasures('pos', refpos, testpos)
>>> printmeasures('neg', refneg, testneg)
```

This gives us more information into the performance of the model for each label.

Cross-validation

As a final topic in evaluation, we have discussed that our testing of the features on the movie review 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 could change the function to report precision and recall for each label.

Exercise

There is no real exercise for this week. Instead post into the discussion group the results of the cross validation that you ran. Since each of us has a different random shuffle, we will particularly look at the mean accuracy for the whole class.