NLP Lab Session
Week 9

Classification and Feature Sets in the NLTK, Part 1

Getting Started

For this lab session download the examples: LabWeek9classifynames.txt and put it in your class folder for copy/pasting examples. Start your Python interpreter session.

```
>>> import nltk
```

These examples and others appear in Chapter 6 of the NLTK book.

The goal of this week's lab is to show how to set up a text classification problems in the NLTK.

Learning to Classify Text

We have seen that many NLP tasks, particularly the semantic tasks such as WSD (Word Sense Disambiguation) and SRL (Semantic Role Labeling) are typically solved used machine learning techniques to classify text. The WSD classification is to take each word in text and label it with one of the word senses from WordNet. The SRL problem is to take each verb in a sentence and label each constituent phrase in the sentence with a semantic role label or not, thus finding phrases in the sentence that can be labeled with one of the semantic roles of that verb.

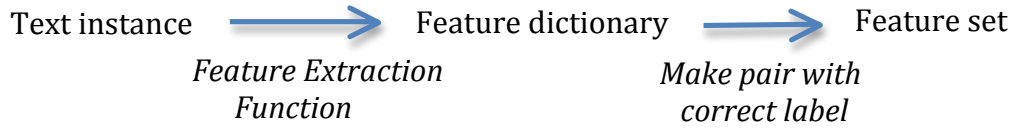
For these types of classification problems, we need to represent the input text by a set of features for the classifier algorithm. Now you may have seen in some machine learning tools such as Weka or SKLearn that there are text processing functions – Weka's is called StringToWordVector – that can obtain a set of features that represent a piece of text by the frequencies of the words that it contains. These functions may also have filters such as stop words or minimum frequencies. But with more advanced text processing, we can define many more types of features from text that are required for some text classification tasks. In our labs, we will look at several ways to define text features for classification.

In this first lab using NLP for classification, we look at how to prepare data for classification in the NLTK. These examples and others appear in Chapter 6 of the NLTK book. For each example, or instance, of the classification problem, we prepare a set of features that will represent that example to the machine learning algorithm. (Look at the diagram in Section 1.)

Name Gender Classifier

We start with a simple problem that will illustrate the process of preparing text data for classification and training and using classifiers. This problem is based on the idea that male and female first names (in English) have distinctive characteristics. For example, names ending in a, e, and i are likely to be female, while names ending in k, o, r, s and t are likely to be male. We will build a classifier that will label any name with its gender.

For each item to be classified, in this case a single word, in NLTK we build the features of that item as a dictionary that maps each feature name to a value, which can be a Boolean, a number or a string. A feature set is the feature dictionary together with the label of the item to be classified, in this case the gender.



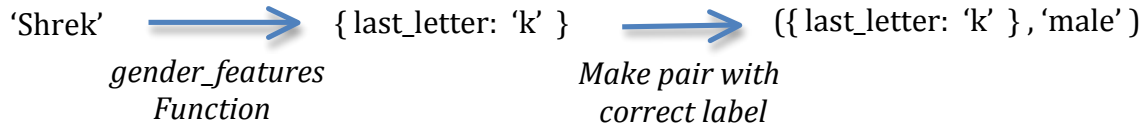
We first define a function that will extract or build the features for a single instance of the problem, in this case a single name. To start with, we will generate a single feature which consists of the last letter of the name. Note that the function returns a dictionary with a single item.

```
>>> def gender_features(word):
    return{'last_letter': word[-1]}
```

We can apply this function to any name:

```
>>> gender_features('Shrek')
```

Now that we've defined our features, we need to construct the training data, or "gold standard" data. This will be a list of first names, each of which will be labeled either male or female. So we want a list of names with known gender where we can construct the feature set for each name. For example, if the gender of Shrek is known to be "male", we could have:



The NLTK corpus contains a names corpus which has a list of male first names and another list of female first names, so we can use this data to create a list of all the first names, but where each is labeled with its gender.

```
>>> from nltk.corpus import names
```

The names corpus has a function words that will return either the names identified by the string 'male.txt' or 'female.txt'. Here we look at the first 20 male names.

```
>>> names.words('male.txt')[:20]
```

From the male and female names lists, we will create one long list with (name, gender) pairs to create the labeled data.

```
>>> namesgender = ([(name, 'male') for name in names.words('male.txt')] +
        [(name, 'female') for name in names.words('female.txt')])
```

Take a look at this list with the first 20 names and the last 20 names.

```
>>> len(namesgender)
7944
>>> namesgender[:20]
>>> namesgender[7924:]
```

Now we create a random shuffle of the namesgender list so that we can easily split it into a training and test set.

```
>>> import random
>>> random.shuffle(namesgender)
>>> namesgender[:20]
```

Next we use the feature extractor function to create the list of instances of the problem that consists only of the features and the gender label. (This is the equivalent of the spread sheet for structured data in a classification problem.) But this example is unusual in that most of the actual text is ignored in the features!

```
>>> featuresets = [(gender_features(n), g) for (n,g) in namesgender]
>>> featuresets[:20]
```

We split this list into training and test sets and run the Naïve Bayes classifier algorithm to create a trained classifier. (The training set is the last 7444 examples and the test set is the first 500 examples.) What do you think will happen if some of the examples with last letter = "n" are female and some of them are male?

```
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

We can the compute the accuracy of the classifier on the test set. The accuracy function for each classifier first removes the gender labels from the test set and runs the classifier on each name in the test set to get a predicted gender. Then it compares the predicted gender with each actual gender from the test set to get the evaluation score. In this case, it just produces an accuracy score, instead of precision and recall.

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

One of the things that we can use a classifier for is to label totally new instances of the problem, in this case, names that come from the future:

```
>>> classifier.classify(gender_features('Neo'))
>>> classifier.classify(gender_features('Trinity'))
```

Finally, the classifier class for Naïve Bayes has a function that shows the feature values that were most important in doing the classification.

```
>>> classifier.show_most_informative_features(20)
```

When the output of this function shows a line like:

```
last_letter = 'a'           female : male   =     35.4 : 1.0
```

it means that when an example has the feature last_letter = ‘a’, it is 35.4 times more likely to be classified as female than male.

Choosing Good Features

Selecting relevant features can usually be the most important part of training a classifier. Often the approach is to throw in as many features as possible and then try to figure out which ones were important. For most machine learning algorithms, throwing in too many features can cause a problem known as “overfitting”, which is that the classifier is trained on so many of the exact details of the training set that it is not as good on new examples.

For the name gender problem, we will try a second feature extraction function that has the first letter, the last letter, a count of each letter, and the individual letters of the name.

[Note that the string “count{}” uses the new string format function of Python3. In this case, it is inserting the value of the variable *letter* into the string that is the name of the feature. For more details, see https://docs.python.org/3.4/library/string.html]

```
>>> def gender_features2(name):
        features = {}
        features["firstletter"] = name[0].lower()
        features["lastletter"] = name[-1].lower()
        for letter in 'abcdefghijklmnopqrstuvwxyz':
                features["count({})".format(letter)] = name.lower().count(letter)
                features["has({})".format(letter)] = (letter in name.lower())
        return features

>>> features = gender_features2('Shrek')
>>> len(features)
>>> features
```

Create new feature sets for all our names in the namesgender list.

```
>>> featuresets2 = [(gender_features2(n), g) for (n,g) in namesgender]
```

We can just look at the feature sets, but it’s more informative to print the name as well:

```
>>>  for (n,g) in namesgender[:3]:
   print (n, gender_features2(n), '\n')
```

We create a new training and test set based on these features. Depending on your random shuffle, you may get a lower result, due to overfitting the very specific features on the training set.

```
>>> train_set, test_set = featuresets2[500:], featuresets2[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

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

In developing a classifier for a classification problem, we will want to do some error analysis of the test set and then perhaps change our features and retrain the classifier. This is known as the development process. In the real world, it would be important to keep a separate test set that was not used in the error analysis for our final evaluation. So we would actually divide our labeled data into a training set, a development test set, and a test set. But for this lab, we will continue to just use a training and a test set.

Setting up Lab Exercise

Going back to using our features with just the last letter of each name, instead of separating the feature sets into training and test, we separate the names into training and test so that we can repeat experiments on the same training set and test set.

```
>>> train_names = namesgender[500:]
>>> test_names = namesgender[:500]

>>> train_set = [(gender_features(n), g) for (n,g) in train_names]
>>> test_set = [(gender_features(n), g) for (n,g) in test_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
```

(Save this classifier accuracy number on the development test set to use for comparison in today's exercise.)

Next we define a function that will get a list of errors by running the classifier on the development test names and comparing it with the original name gender labels (comparing the classifier labels with the gold standard labels).

```
>>> def geterrors(test):
    errors = []
    for (name, tag) in test:
        guess = classifier.classify(gender_features(name))
        if guess != tag:
            errors.append( (tag, guess, name) )
    return errors

>>> errors = geterrors(test_names)
>>> len(errors)
```

Then we define a function to print all the errors, sorted by the correct labels, so that we can look at the differences (with even more string formatting).

```
>>> def printerrors(errors):
    for (tag, guess, name) in sorted(errors):
        print('correct={:<8s} guess={:<8s} name={:<30s}'.format(tag, guess, name))
>>> printerrors(errors)
```

Looking through the list of errors, we observe cases where using the last two letters of each name might be more informative. As observed in the book, "For example, names ending in *yn* appear to be predominantly female, despite the fact that names ending in *n* tend to be male; and names ending in *ch* are usually male, even though names that end in *h* tend to be female."

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

```
>>> cm = nltk.metrics.ConfusionMatrix(reflist, testlist)
>>> print (cm)
      |   f     |
      |   e     |
      |   m    m |
      |   a    a |
      |   l    l |
      |   e    e |
-------+---------+
female |<260> 63 |
  male |  58<119>|
-------+---------+
(row = reference; col = test)
```

In our gender name classification task, we have two class labels: ‘female’ and ‘male’ (instead of Yes and No). If we consider the female class as the positive class and the male as the negative class, there are 260 True Positives, 119 True Negatives, 58 False Positives and 63 False Negatives. Since this classification task is symmetric with respect to the two classes, we can flip the terminology and consider the male class as

positive and the female class as negative. In that case, there are 119 True Positives, 260 True Negatives, 63 False Positives, and 58 False Negatives.

We could now just use some arithmetic to compute the precision and recall, but the NLTK has functions to do this. Unfortunately, these functions require a different setup than the confusion matrix function. 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. Here we set up reference and test sets for each label that use the index number as the item identifiers.

```
>>> reffemale = set([i for i,label in enumerate(reflist) if label == 'female'])
>>> refmale = set([i for i,label in enumerate(reflist) if label == 'male'])
>>> testfemale = set([i for i,label in enumerate(testlist) if label == 'female'])
>>> testmale = set([i for i,label in enumerate(testlist) if label == 'male'])
>>> reffemale
>>> testfemale
>>> refmale
>>> testmale
```

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('female', reffemale, testfemale)
>>> printmeasures('male', refmale, testmale)
```

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

If we then want the average precision and recall, NLTK has functions that compute that from the sets of our combined reference and test lists.

Exercise:

Define a new feature extraction function that includes features for two-letter suffixes, such as the one here:

```
>>> def gender_features3(word):
   return {‘suffix1’: word[-1],
           ‘suffix2’: word[-2]}
```

Keep the variables train_names and test_names that define the training and test set.

Make new train_set and test_set variables. Carry out the classification and look at the errors in the test set.

Is this classification more accurate than the baseline with one suffix? Can you see error examples that you could use to make new features to improve? (You don't have to do this, just observe it.) How does it compare with the result with the 54 features?

Make a post in the discussion for the Week 9 lab in the blackboard system with your original accuracy on the test set and the new accuracy, and you may also make any observations that you can about the remaining errors.

If you have time, you can make a new gender_features function that keeps three suffix letters, but make allowances if any names are only 2 characters long. Or perhaps a gender_features function that uses the first letter and the last two letters. What about a feature that is the count of the number of letters, do you think that would help?