Lab 9: Supervised Text Classification and Evaluation (Part 2)

**References:**

1. Regular Expressions: The Complete Tutorial, by Jan Goyvaerts, 2007.
2. Speech and Language Processing, by Dan Jurafsky and James H. Martin. Prentice Hall Series in Artificial Intelligence, 2008.
3. Natural Language Processing with Python, by Steven Bird, Ewan Klein and Edward Loper, 2014.

Quick Review

Classification is the task of choosing the correct class label for a given input. In basic classification tasks, each input is considered in isolation from all other inputs, and the set of labels is defined in advance.

In order to decide whether a classification model is accurately capturing a pattern, we must evaluate that model. The result of this evaluation is important for deciding how trustworthy the model is, and for what purposes we can use it. Evaluation can also be an effective tool for guiding us in making future improvements to the model. The simplest metric that can be used to evaluate a classifier, accuracy, measures the percentage of inputs in the test set that the classifier correctly labelled.

Practices

Choosing the Right Features and Accuracy

Selecting relevant features and deciding how to encode them for a learning method can have an enormous impact on the learning method's ability to extract a good model. Much of the interesting work in building a classifier is deciding what features might be relevant, and how we can represent them. Although it's often possible to get decent performance by using a fairly simple and obvious set of features, there are usually significant gains to be had by using carefully constructed features based on a thorough understanding of the task at hand.

Typically, feature extractors are built through a process of trial-and-error, guided by intuitions about what information is relevant to the problem. It's common to start with a "kitchen sink" approach, including all the features that you can think of, and then checking to see which features actually are helpful. We take this approach for name gender features in 1.2.

def gender\_features2(name):

features = {}

features["first\_letter"] = name[0].lower()

features["last\_letter"] = 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

By adding the following command, you can see the feature set for a name:

gender\_features2('John')

However, there are usually limits to the number of features that you should use with a given learning algorithm — if you provide too many features, then the algorithm will have a higher chance of relying on idiosyncrasies of your training data that don't generalize well to new examples. This problem is known as **overfitting**, and can be especially problematic when working with small training sets. For example, if we train a naive Bayes classifier using the feature extractor above, it will overfit the relatively small training set, resulting in a system whose accuracy is about 1% lower than the accuracy of a classifier that only pays attention to the final letter of each name:

featuresets = [(gender\_features2(n), gender) for (n, gender) in labeled\_names]

train\_set, test\_set = featuresets[500:], featuresets[:500]

classifier = nltk.NaiveBayesClassifier.train(train\_set)

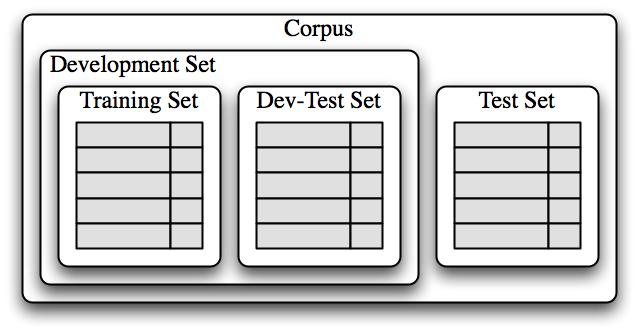
print(nltk.classify.accuracy(classifier, test\_set))

Once an initial set of features has been chosen, a very productive method for refining the feature set is **error analysis**. First, we select a development set, containing the corpus data for creating the model. This development set is then subdivided into the **training set** and the **dev-test set**.

train\_names = labeled\_names[1500:]

devtest\_names = labeled\_names[500:1500]

test\_names = labeled\_names[:500]

The training set is used to train the model, and the dev-test set is used to perform error analysis. The test set serves in our final evaluation of the system. For reasons discussed below, it is important that we employ a separate dev-test set for error analysis, rather than just using the test set. The division of the corpus data into different subsets is shown here.

Having divided the corpus into appropriate datasets, we train a model using the training set, and then run it on the dev-test set.

train\_set = [(gender\_features(n), gender) for (n, gender) in train\_names]

devtest\_set = [(gender\_features(n), gender) for (n, gender) in devtest\_names]

test\_set = [(gender\_features(n), gender) for (n, gender) in test\_names]

classifier = nltk.NaiveBayesClassifier.train(train\_set)

print(nltk.classify.accuracy(classifier, devtest\_set))

Using the dev-test set, we can generate a list of the errors that the classifier makes when predicting name genders:

errors = []

for (name, tag) in devtest\_names:

guess = classifier.classify(gender\_features(name))

if guess != tag:

errors.append((tag, guess, name))

We can then examine individual error cases where the model predicted the wrong label, and try to determine what additional pieces of information would allow it to make the right decision (or which existing pieces of information are tricking it into making the wrong decision). The feature set can then be adjusted accordingly. The names classifier that we have built generates about 100 errors on the dev-test corpus:

for (tag, guess, name) in sorted(errors):

print('correct={:<8} guess={:<8s} name={:<30}'.format(tag, guess, name))

Looking through this list of errors makes it clear that some suffixes that are more than one letter can be indicative of name genders. 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. We therefore adjust our feature extractor to include features for two-letter suffixes:

def gender\_features(word):

return {'suffix1': word[-1:],

'suffix2': word[-2:]}

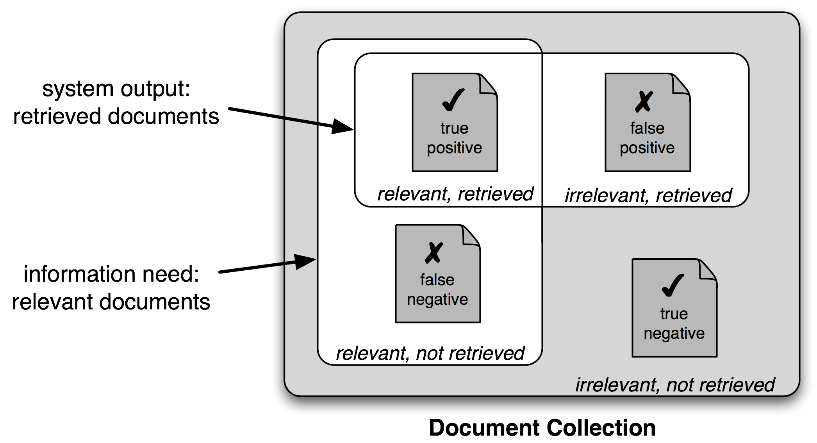
Rebuilding the classifier with the new feature extractor, we see that the performance on the dev-test dataset improves by almost 2 percentage points.

This error analysis procedure can then be repeated, checking for patterns in the errors that are made by the newly improved classifier. Each time the error analysis procedure is repeated, we should select a different dev-test/training split, to ensure that the classifier does not start to reflect idiosyncrasies in the dev-test set.

But once we've used the dev-test set to help us develop the model, we can no longer trust that it will give us an accurate idea of how well the model would perform on new data. It is therefore important to keep the test set separate, and unused, until our model development is complete. At that point, we can use the test set to evaluate how well our model will perform on new input values.

Gender Identification - Other Evaluation Metrics (Precision and Recall)

It is therefore conventional to employ a different set of measures for search tasks, based on the number of items in each of the four categories shown below:

* **True Positives** are relevant items that we correctly identified as relevant.
* **True Negatives** are irrelevant items that we correctly identified as irrelevant.
* **False Positives** (or **Type I errors**) are irrelevant items that we incorrectly identified as relevant.
* **False Negatives** (or **Type II errors**) are relevant items that we incorrectly identified as irrelevant.

Given these four numbers, we can define the following metrics:

* **Precision**, which indicates how many of the items that we identified were relevant, is .
* **Recall**, which indicates how many of the relevant items that we identified, is .
* The **F-Measure** (or **F-Score**), which combines the precision and recall to give a single score, is defined to be the harmonic mean of the precision and recall: .

The following codes provide the evaluation results for gender identification, discussed above.

import nltk

from nltk.corpus import names

import random

def gender\_features(word):

return {'suffix1': word[-1:],

'suffix2': word[-2:]}

labeled\_names = ([(name, 'male') for name in names.words('male.txt')] +

[(name, 'female') for name in names.words('female.txt')])

random.shuffle(labeled\_names)

train\_names = labeled\_names[1500:]

devtest\_names = labeled\_names[500:1500]

test\_names = labeled\_names[:500]

train\_set = [(gender\_features(n), gender) for (n, gender) in train\_names]

devtest\_set = [(gender\_features(n), gender) for (n, gender) in devtest\_names]

test\_set = [(gender\_features(n), gender) for (n, gender) in test\_names]

classifier = nltk.NaiveBayesClassifier.train(train\_set)

print("\nThe accuracy is: ", nltk.classify.accuracy(classifier, devtest\_set)\*100,"%")

errors = []

prediction\_list = []

TP=0

TN=0

FP=0

FN=0

for (name, tag) in devtest\_names:

guess = classifier.classify(gender\_features(name))

prediction\_list.append((tag, guess, name))

if guess == 'male':

if tag == 'male':

TP+=1

else:

FP+=1

else:

if tag == 'female':

TN+=1

else:

FN+=1

if guess != tag:

errors.append((tag, guess, name))

print('total number of test items: ',len(prediction\_list))

print('total number of errors: ', len(errors))

print('TP: ', TP)

print('FP: ', FP)

print('TN: ', TN)

print('FN: ', FN)

Using nltk.ConfusionMatrix() is an easier way to produce the confusion matrix:

def column(matrix, i):

return [row[i] for row in matrix]

cm = nltk.ConfusionMatrix(column(prediction\_list,0), column(prediction\_list,1))

print(cm.pretty\_format(sort\_by\_count=True, show\_percents=True, truncate=9))

Word Vectorisation

Word Embedding or Word vectorization is a methodology in NLP to map words or phrases from vocabulary to a corresponding vector of real numbers which used to find word predictions, word similarities/semantics.

The process of converting words into numbers are called Vectorization.

**TfidfVectorizer** and **CountVectorizer**both are methods for converting text data into vectors as model can process only numerical data.

In **CountVectorizer**we **only count the number of times a word appears in the document**which results in biasing in favour of most frequent words.

sample = ['problem of evil', 'evil queen', 'horizon problem']

print("\nText: ", sample)

print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ")

# Count Vectorizer ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

print("\nCount Vectorizer ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")

from sklearn.feature\_extraction.text import CountVectorizer

count\_vectorizer = CountVectorizer()

# count\_vectorizer = CountVectorizer(lowercase=True, stop\_words='english')

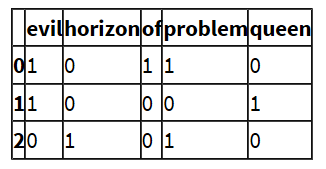
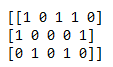
count\_vectorizer.fit(sample)

print('\nVocabulary: \n', count\_vectorizer.vocabulary\_)

X\_train\_counts = count\_vectorizer.fit\_transform(sample).toarray()

print("\n", X\_train\_counts)

**Output: Python Output:**



This ends up in ignoring rare words which could have helped is in processing our data more efficiently.

***To overcome this, we use TfidfVectorizer.***

In **TfidfVectorizer**we consider **overall document weightage** of a word. It helps us in dealing with most frequent words. Using it we can penalize them. TfidfVectorizer weights the word counts by a measure of how often they appear in the documents.

# Tfidf Vectorizer ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

print("\nTfidf Vectorizer ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")

from sklearn.feature\_extraction.text import TfidfVectorizer

TFIDF\_vectorizer = TfidfVectorizer()

# TFIDF\_vectorizer = TfidfVectorizer(lowercase=True, stop\_words='english')

TFIDF\_vectorizer.fit(sample)

print('\nVocabulary: \n', TFIDF\_vectorizer.vocabulary\_)

X = TFIDF\_vectorizer.fit\_transform(sample)

print("\n", X)

**Output: Python Output:**

