CE807 Lab 3
Text classification with Python

In this lab we are going to use scikit-learn for text classification, focusing in particular on the most classic example of text classification: spam detection. The lab is roughly based on Zac Stewart's tutorial at:

http://zacstewart.com/2015/04/28/document-classification-with-scikit-learn.html

Make sure that all material in the zip file provided, including the `data` sub-folder, is in a directory in your file space.

1. Datasets

There are several publicly available datasets to study spam classification, such as

Enron-Spam: http://www.aueb.gr/users/ion/data/enron-spam/

SpamAssassin: https://spamassassin.apache.org/publiccorpus/

We are going to use SpamAssassin here – you can find a version in the data/ subdirectory. Download the zipped archive `data.zip` in your folder and unzip it.

Exercise: read the description of the corpus at:
https://spamassassin.apache.org/publiccorpus/readme.html

2. Training and testing a spam detector

The steps involved in developing a classifier for a task are:

1. Putting the data in a format that the learning platform (in this case scikit-learn) can use
2. Decide which features we want to use, and develop methods to extract them
3. Training a classifier
4. Evaluating the classifier
5. Improve its performance (e.g., by adding more features or by trying a different ML algorithm)

The script classifier.py that you can find in the Lab3 directory carries out these four steps. We are going to discuss each step in turn.

2.1 Putting the data in the right format

Scikit-learn likes data to be in a Numpy-style array. A suitable representation for the data to train a classifier would be an array with two columns: the text of the emails in one column, and the class in a second column.

This is the task of the function `build_data_frame`, that takes as input a `(directory, classification)` pair from the `SOURCES` variable, uses `read_files` to read the content of each file in the directory, and adds for each of those files a (text, classification) pair to the data frame.

2.2 Feature identification and extraction

The next step is to identify the features that we want to use to categorize documents, and to extract them from each document in the collection.

In our case, we're going to use the words contained in the email messages as features. And we're going to use once again the `CountVectorizer` class from scikit-learn's `feature_extraction` module which we have used in the previous two labs –specifically, the `fit_transform` function—to simultaneously learn the vocabulary of the corpus and extract word count features from each message (Please note: first you need to properly set up a `DataFrame` and store it in the variable `data`, see accompanying `classifier.py`).

```
import numpy
from sklearn.feature_extraction.text import CountVectorizer

count_vectorizer = CountVectorizer()
counts = count_vectorizer.fit_transform(data['text'].values)
```

(In fact, as we will see below, scikit-learn makes it possible to combine feature extraction with training using **pipelines**.)

2.3 Training a Naïve Bayes classifier

After having extracted the featural vector representations (= word counts) from documents, we can use these to train a model. Specifically, we're going to train a NaiveBayes classifier like discussed in the lectures.

Scikit-learn includes implementations of different types of Naïve Bayes classifiers; we're going to use here MultinomialNB, described at:

http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html
The MultinomialNB implementation of Naïve Bayes provides two main functions:

- `fit`, which takes as input a set of feature vectors and outputs a model;
- `predict`, which given a model, specifies the class a new document belongs to.

Exercise study the documentation for MultinomialNB, and compare it with that of KMeans discussed in Lab 2. That page also contains a short example of how to use it.

Fit can be used to train a Naïve Bayes model using the document vectors as follows:

```
from sklearn.naive_bayes import MultinomialNB
classifier = MultinomialNB()
targets = data['class'].values
classifier.fit(counts, targets)
```

Having trained a classifier, we can use it to predict the class of a document as follows:

```
examples = ['Free Viagra call today!', "I'm going to attend the Linux
users group tomorrow."]
example_counts = count_vectorizer.transform(examples)
predictions = classifier.predict(example_counts)
predictions # [1, 0]
```

2.4 Setting up a pipeline

In fact, the two tasks of feature extraction and model fitting can be combined by defining a **pipeline** that, taken as input a set of documents, uses them to train a model.

This can be done using the Pipeline class described at:

http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

Exercise study the documentation for the class

The following code specifies a Pipeline using CountVectorizer to vectorize and of MultinomialNB to train:

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([
          ('vectorizer',  CountVectorizer()),
          ('classifier',  MultinomialNB()) ])
pipeline.fit(data['text'].values, data['class'].values)
pipeline.predict(examples) # ['spam', 'ham']
```

2.5 Cross-validation

Finally, we will see how to use **cross-validation** to assess the performance of the model.

k-fold cross validation is a way for testing that involves splitting your dataset into k **folds**, then in turn use each of the k folds as a test set, and using the other k-1 to train.

Scikit-learn makes it very easy to do cross-validation: the class KFold takes as argument the size of the dataset and the number of folds, and generates k pairs of index vectors, one for each of the k iterations.

```
from sklearn.cross_validation import KFold
from sklearn.metrics import confusion_matrix, f1_score

k_fold = KFold(n=len(data), n_folds=6)
scores = []
confusion = numpy.array([[0, 0], [0, 0]])

for train_indices, test_indices in k_fold:
     train_text = data.iloc[train_indices]['text'].values
     train_y = data.iloc[train_indices]['class'].values

     test_text = data.iloc[test_indices]['text'].values
     test_y = data.iloc[test_indices]['class'].values

     pipeline.fit(train_text, train_y)
     predictions = pipeline.predict(test_text)
     confusion += confusion_matrix(test_y, predictions)
     score = f1_score(test_y, predictions, pos_label=SPAM)
     scores.append(score)

print('Total emails classified:', len(data))
print('Score:', sum(scores)/len(scores))
print('Confusion matrix:')
print(confusion)
# Total emails classified: 55326
# Score: 0.942661080942
# Confusion matrix: # [[21660   178] #  [ 3473 30015]]
```

2.6 Putting it all together

The script classifier.py does all the steps just discussed.

Exercise analyse the script and make sure you understand what it's doing, then run it.

3. Improving the performance of the first spam detector

In this part of the lab, your task will be to try and improve the performance of the spam detector developed in Section 2. The idea is that you should try to modify the original script, and test whether the new program is better than the previous one. (This is the task that you will have to do in Assignment 1!) Here are three suggestions, but if you think of something else., go ahead.

3.1 Using a different classifier

Within a platform like scikit-learn, the easiest way to try and improve a system is by trying different machine learning algorithms and/or different parameters for them.

The most minimal modification is to try to use a different type of Naïve Bayes classifier—eg., the Bernoulli Naïve Bayes classifier, that takes as input a vector of binary values rather than a vector of counts.

Exercise learn about `BernoulliNB` in the scikit-learn documentation.

To change to using this classifier, all you have to do is to replace the part of the code in the script that adds a NaiveBayes classifier to the Pipeline object. `BernoulliNB` has a `binarize` parameter that sets the threshold for converting numeric values to booleans-If we set that to 0.0 we will convert words which are not present to `False` and words which are to `True`.

```
from sklearn.naive_bayes import BernoulliNB

pipeline = Pipeline([
     ('count_vectorizer',   CountVectorizer(ngram_range=(1, 2))),
     ('classifier',         BernoulliNB(binarize=0.0)) ])
```

Exercise modify the classifier.py script to use `BernoulliNB`, and evaluate the results. Do you see an improvement?

Exercise try at least one additional type of machine learning algorithm, and again, try to see if you can improve the performance of your spam detector. For instance, you could try to use Support Vector Machines:

http://scikit-learn.org/stable/modules/svm.html

Exercise try using a grid search to optimise the machine learning algorithm:

1) http://scikit-learn.org/stable/modules/grid_search.html
2) Example:http://scikit-learn.org/stable/auto_examples/model_selection/grid_search_text_feature_extraction.html#sphx-glr-auto-examples-model-selection-grid-search-text-feature-extraction-py

3.2 Using n-grams

A second option is to use as features n-grams of length greater than 1 instead of just unigrams (i.e., tokens). For instance, the bigram 'lose weight' is most likely a better feature than either 'lose' or 'weight'.

`CountVectorizer` automatically computes n-grams for you. You can include bigrams and other n-grams among your features by specifying an appropriate value for the `ngram_range` parameter.

```
pipeline = Pipeline([
```

```
('count_vectorizer', CountVectorizer(ngram_range=(1, 2))),
('classifier',       MultinomialNB()) ])
```

Exercise learn about ngram_range.
Exercise try different ranges of n-grams. Does performance improve? What about the time it takes to train?

3.3 Using other feature selection mechanisms

Finally, you could try to use other vectorizers –e.g., StemVectorizer that we developed in Lab1, or StemmedTfIdfVectorizer that we used in Lab2, etc.

Exercise Try other vectorizers, and again, compare your results with those you obtained before.

References

- scikit-learn's documentation at http://scikit-learn.org/stable/index.html
- Zac Stewart's tutorial on spam detection with scikit learn at: http://zacstewart.com/2015/04/28/document-classification-with-scikit-learn.html