Working With Text Data

The goal of this guide is to explore some of the main `scikit-learn` tools on a single practical task: analysing a collection of text documents (newsgroups posts) on twenty different topics.

In this section we will see how to:

- load the file contents and the categories
- extract feature vectors suitable for machine learning
- train a linear model to perform categorization
- use a grid search strategy to find a good configuration of both the feature extraction components and the classifier

Tutorial setup

To get started with this tutorial, you firstly must have the *scikit-learn* and all of its required dependencies installed.

Please refer to the *installation instructions* page for more information and for per-system instructions.

The source of this tutorial can be found within your scikit-learn folder:

```
scikit-learn/doc/tutorial/text_analytics/
```

The tutorial folder, should contain the following folders:

- `*.rst files` - the source of the tutorial document written with sphinx
- `data` - folder to put the datasets used during the tutorial
- `skeletons` - sample incomplete scripts for the exercises
- `solutions` - solutions of the exercises

You can already copy the skeletons into a new folder somewhere on your hard-drive named `sklearn_tut_workspace` where you will edit your own files for the exercises while keeping the original skeletons intact:

```
% cp -r skeletons work_directory/sklearn_tut_workspace
```

Machine Learning algorithms need data. Go to each `$TUTORIAL_HOME/data` sub-folder and run the `fetch_data.py` script from there (after having read them first).

For instance:

```
% cd $TUTORIAL_HOME/data/languages
% less fetch_data.py
% python fetch_data.py
```

Loading the 20 newsgroups dataset

The dataset is called “Twenty Newsgroups”. Here is the official description, quoted from the website:

> The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly

across 20 different newsgroups. To the best of our knowledge, it was originally collected by Ken Lang, probably for his paper “Newsweeder: Learning to filter netnews,” though he does not explicitly mention this collection. The 20 newsgroups collection has become a popular data set for experiments in text applications of machine learning techniques, such as text classification and text clustering.

In the following we will use the built-in dataset loader for 20 newsgroups from scikit-learn. Alternatively, it is possible to download the dataset manually from the web-site and use the `sklearn.datasets.load_files` function by pointing it to the `20news-bydate-train` subfolder of the uncompressed archive folder.

In order to get faster execution times for this first example we will work on a partial dataset with only 4 categories out of the 20 available in the dataset:

```
>>> categories = ['alt.atheism', 'soc.religion.christian',
...               'comp.graphics', 'sci.med']
```

We can now load the list of files matching those categories as follows:

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> twenty_train = fetch_20newsgroups(subset='train',
...     categories=categories, shuffle=True, random_state=42)
```

The returned dataset is a `scikit-learn` “bunch”: a simple holder object with fields that can be both accessed as python `dict` keys or `object` attributes for convenience, for instance the `target_names` holds the list of the requested category names:

```
>>> twenty_train.target_names
['alt.atheism', 'comp.graphics', 'sci.med', 'soc.religion.christ
```

The files themselves are loaded in memory in the `data` attribute. For reference the filenames are also available:

```
>>> len(twenty_train.data)
2257
>>> len(twenty_train.filenames)
2257
```

Let's print the first lines of the first loaded file:

```
>>> print("\n".join(twenty_train.data[0].split("\n")[:3]))
From: sd345@city.ac.uk (Michael Collier)
Subject: Converting images to HP LaserJet III?
Nntp-Posting-Host: hampton

>>> print(twenty_train.target_names[twenty_train.target[0]])
comp.graphics
```

Supervised learning algorithms will require a category label for each document in the training set. In this case the category is the name of the newsgroup which also happens to be the name of the folder holding the individual documents.

For speed and space efficiency reasons `scikit-learn` loads the target attribute as an array of integers that corresponds to the index of the category name in the `target_names` list. The category integer id of each sample is stored in the `target` attribute:

```
>>> twenty_train.target[:10]
array([1, 1, 3, 3, 3, 3, 3, 2, 2, 2])
```

It is possible to get back the category names as follows:

```
>>> for t in twenty_train.target[:10]:
...     print(twenty_train.target_names[t])
...
comp.graphics
comp.graphics
soc.religion.christian
soc.religion.christian
soc.religion.christian
soc.religion.christian
soc.religion.christian
sci.med
sci.med
sci.med
```

You can notice that the samples have been shuffled randomly (with a fixed RNG seed): this is useful if you select only the first samples to quickly train a model and get a first idea of the results before re-training on the complete dataset later.

Extracting features from text files

In order to perform machine learning on text documents, we first need to turn the text content into numerical feature vectors.

Bags of words

The most intuitive way to do so is the bags of words representation:

1. assign a fixed integer id to each word occurring in any document of the training set (for instance by building a dictionary from words to integer indices).
2. for each document `#i`, count the number of occurrences of each word `w` and store it in `X[i, j]` as the value of feature `#j` where `j` is the index of word `w` in the dictionary

The bags of words representation implies that `n_features` is the number of distinct words in the corpus: this number is typically larger that 100,000.

If `n_samples == 10000`, storing `X` as a numpy array of type float32 would require 10000 x 100000 x 4 bytes = **4GB in RAM** which is barely manageable on today's computers.

Fortunately, **most values in X will be zeros** since for a given document less than a couple thousands of distinct words will be used. For this reason we say that bags of words are typically **high-dimensional sparse datasets**. We can save a lot of memory by only storing the non-zero parts of the feature vectors in memory.

`scipy.sparse` matrices are data structures that do exactly this, and `scikit-learn` has built-in support for these structures.

Tokenizing text with `scikit-learn`

Text preprocessing, tokenizing and filtering of stopwords are included in a high level component that is able to build a dictionary of features and transform documents to feature vectors:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count_vect = CountVectorizer()
>>> X_train_counts = count_vect.fit_transform(twenty_train.data)
>>> X_train_counts.shape
(2257, 35788)
```

`CountVectorizer` supports counts of N-grams of words or consequective characters. Once fitted, the vectorizer has built a dictionary of feature indices:

```
>>> count_vect.vocabulary_.get(u'algorithm')
4690
```

The index value of a word in the vocabulary is linked to its frequency in the whole training corpus.

From occurrences to frequencies

Occurrence count is a good start but there is an issue: longer documents will have higher average count values than shorter documents, even though they might talk about the same topics.

To avoid these potential discrepancies it suffices to divide the number of occurrences of each word in a document by the total number of words in the document: these new features are called `tf` for Term Frequencies.

Another refinement on top of tf is to downscale weights for words that occur in many documents in the corpus and are therefore less informative than those that occur only in a smaller portion of the corpus.

This downscaling is called tf–idf for “Term Frequency times Inverse Document Frequency”.

Both **tf** and **tf–idf** can be computed as follows:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tf_transformer = TfidfTransformer(use_idf=False).fit(X_trair
>>> X_train_tf = tf_transformer.transform(X_train_counts)
>>> X_train_tf.shape
(2257, 35788)
```

In the above example-code, we firstly use the `fit(..)` method to fit our estimator to the data and secondly the `transform(..)` method to transform our count-matrix to a tf-idf representation. These two steps can be combined to achieve the same end result faster by skipping redundant processing. This is done through using the `fit_transform(..)` method as shown below, and as mentioned in the note in the previous section:

```
>>> tfidf_transformer = TfidfTransformer()
>>> X_train_tfidf = tfidf_transformer.fit_transform(X_train_cour
>>> X_train_tfidf.shape
(2257, 35788)
```

Training a classifier

Now that we have our features, we can train a classifier to try to predict the category of a post. Let’s start with a *naïve Bayes* classifier, which provides a nice baseline for this task. `scikit-learn` includes several variants of this classifier; the one most suitable for word counts is the multinomial variant:

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB().fit(X_train_tfidf, twenty_train.target
```

To try to predict the outcome on a new document we need to extract the features using almost the same feature extracting chain as before. The difference is that we call `transform` instead of `fit_transform` on the transformers, since they have already been fit to the training set:

```
>>> docs_new = ['God is love', 'OpenGL on the GPU is fast']
>>> X_new_counts = count_vect.transform(docs_new)
>>> X_new_tfidf = tfidf_transformer.transform(X_new_counts)

>>> predicted = clf.predict(X_new_tfidf)

>>> for doc, category in zip(docs_new, predicted):
...     print('%r => %s' % (doc, twenty_train.target_names[categ
...
'God is love' => soc.religion.christian
'OpenGL on the GPU is fast' => comp.graphics
```

Building a pipeline

In order to make the vectorizer => transformer => classifier easier to work with, `scikit-learn` provides a `Pipeline` class that behaves like a compound classifier:

```
>>> from sklearn.pipeline import Pipeline
>>> text_clf = Pipeline([('vect', CountVectorizer()),
...                      ('tfidf', TfidfTransformer()),
...                      ('clf', MultinomialNB()),
... ])
```

The names `vect`, `tfidf` and `clf` (classifier) are arbitrary. We shall see their use in the section on grid search, below. We can now train the model with a single command:

```
>>> text_clf = text_clf.fit(twenty_train.data, twenty_train.targ
```

Evaluation of the performance on the test

set

Evaluating the predictive accuracy of the model is equally easy:

```
>>> import numpy as np
>>> twenty_test = fetch_20newsgroups(subset='test',
...     categories=categories, shuffle=True, random_state=42)
>>> docs_test = twenty_test.data
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.834...
```

I.e., we achieved 83.4% accuracy. Let's see if we can do better with a linear *support vector machine (SVM)*, which is widely regarded as one of the best text classification algorithms (although it's also a bit slower than naïve Bayes). We can change the learner by just plugging a different classifier object into our pipeline:

```
>>> from sklearn.linear_model import SGDClassifier
>>> text_clf = Pipeline([('vect', CountVectorizer()),
...                      ('tfidf', TfidfTransformer()),
...                      ('clf', SGDClassifier(loss='hinge', per
...                                            alpha=1e-3, n_ite
... ])
>>> _ = text_clf.fit(twenty_train.data, twenty_train.target)
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.912...
```

`scikit-learn` further provides utilities for more detailed performance analysis of the results:

```
>>> from sklearn import metrics
>>> print(metrics.classification_report(twenty_test.target, prec
...     target_names=twenty_test.target_names))
...
```

```
                        precision    recall  f1-score   support

           alt.atheism       0.95      0.81      0.87       319
         comp.graphics       0.88      0.97      0.92       389
               sci.med       0.94      0.90      0.92       396
soc.religion.christian       0.90      0.95      0.93       398

           avg / total       0.92      0.91      0.91      1502


>>> metrics.confusion_matrix(twenty_test.target, predicted)
array([[258,  11,  15,  35],
       [  4, 379,   3,   3],
       [  5,  33, 355,   3],
       [  5,  10,   4, 379]])
```

As expected the confusion matrix shows that posts from the newsgroups on atheism and christian are more often confused for one another than with computer graphics.

Parameter tuning using grid search

We've already encountered some parameters such as `use_idf` in the `TfidfTransformer`. Classifiers tend to have many parameters as well; e.g., `MultinomialNB` includes a smoothing parameter `alpha` and `SGDClassifier` has a penalty parameter `alpha` and configurable loss and penalty terms in the objective function (see the module documentation, or use the Python `help` function, to get a description of these).

Instead of tweaking the parameters of the various components of the chain, it is possible to run an exhaustive search of the best parameters on a grid of possible values. We try out all classifiers on either words or bigrams, with or without idf, and with a penalty parameter of either 0.01 or 0.001 for the linear SVM:

```
>>> from sklearn.grid_search import GridSearchCV
>>> parameters = {'vect__ngram_range': [(1, 1), (1, 2)],
...               'tfidf__use_idf': (True, False),
...               'clf__alpha': (1e-2, 1e-3),
... }
```

Obviously, such an exhaustive search can be expensive. If we have multiple CPU cores at our disposal, we can tell the grid searcher to try these eight parameter combinations in parallel with the `n_jobs` parameter. If we give this parameter a value of `-1`, grid search will detect how many cores are installed and uses them all:

```
>>> gs_clf = GridSearchCV(text_clf, parameters, n_jobs=-1)
```

The grid search instance behaves like a normal `scikit-learn` model. Let's perform the search on a smaller subset of the training data to speed up the computation:

```
>>> gs_clf = gs_clf.fit(twenty_train.data[:400], twenty_train.ta
```

The result of calling `fit` on a `GridSearchCV` object is a classifier that we can use to `predict`:

```
>>> twenty_train.target_names[gs_clf.predict(['God is love'])]
'soc.religion.christian'
```

but otherwise, it's a pretty large and clumsy object. We can, however, get the optimal parameters out by inspecting the object's `grid_scores_` attribute, which is a list of parameters/score pairs. To get the best scoring attributes, we can do:

```
>>> best_parameters, score, _ = max(gs_clf.grid_scores_, key=lam
>>> for param_name in sorted(parameters.keys()):
...     print("%s: %r" % (param_name, best_parameters[param_name
...
clf__alpha: 0.001
tfidf__use_idf: True
vect__ngram_range: (1, 1)

>>> score
0.900...
```

Exercises

To do the exercises, copy the content of the ‘skeletons’ folder as a new folder named ‘workspace’:

```
% cp -r skeletons workspace
```

You can then edit the content of the workspace without fear of loosing the original exercise instructions.

Then fire an ipython shell and run the work-in-progress script with:

```
[1] %run workspace/exercise_XX_script.py arg1 arg2 arg3
```

If an exception is triggered, use `%debug` to fire-up a post mortem ipdb session.

Refine the implementation and iterate until the exercise is solved.

For each exercise, the skeleton file provides all the necessary import statements, boilerplate code to load the data and sample

code to evaluate the predictive accurracy of the model.

Exercise 1: Language identification

- Write a text classification pipeline using a custom preprocessor and `CharNGramAnalyzer` using data from Wikipedia articles as training set.
- Evaluate the performance on some held out test set.

ipython command line:

```
%run workspace/exercise_01_language_train_model.py data/language
```

Exercise 2: Sentiment Analysis on movie reviews

- Write a text classification pipeline to classify movie reviews as either positive or negative.
- Find a good set of parameters using grid search.
- Evaluate the performance on a held out test set.

ipython command line:

```
%run workspace/exercise_02_sentiment.py data/movie_reviews/txt_s
```

Exercise 3: CLI text classification utility

Using the results of the previous exercises and the `cPickle` module of the standard library, write a command line utility that detects the language of some text provided on `stdin` and estimate the polarity (positive or negative) if the text is written in English.

Bonus point if the utility is able to give a confidence level for its predictions.

Where to from here

Here are a few suggestions to help further your scikit-learn intuition upon the completion of this tutorial:

- Try playing around with the `analyzer` and `token normalisation` under **`CountVectorizer`**
- If you don't have labels, try using *Clustering* on your problem.
- If you have multiple labels per document, e.g categories, have a look at the *Multiclass and multilabel section*
- Try using *Truncated SVD* for latent semantic analysis.
- Have a look at using *Out-of-core Classification* to learn from data that would not fit into the computer main memory.
- Have a look at the *Hashing Vectorizer* as a memory efficient alternative to **`CountVectorizer`**.