DISCnet Machine Learning Course

Hands-on Machine Learning 101

Introduction and Acknowledgements

This tutorial is largely based on the official "Working With Text Data" scikit-learn tutorial. A number of changes have been made to ensure that it fits the format of our machine learning course, and we've added additional bits that demonstrate how to cluster data with k-means which will give a greater understanding of the datasets we're using.

Through this tutorial you'll learn how to:

- Load a dataset of newsgroup posts
- Extract feature vectors suitable for machine learning
- Use K-Means clustering to automatically group the data
- Train a classifier to that predicts what newsgroup a post belongs to
- Use a grid search strategy to find a good configuration of both the feature extraction components and the classifier

In the follow-up exercises you'll apply what you've learned to two different problems on different datasets.

Prerequisites

To use this tutorial you'll use the Python 3 language with the `scikit-learn` package to perform feature extraction, clustering and classification tasks. `scikit-learn` is an open-source package containing a number of simple and efficient tools for data mining and data analysis in the Python language.

You'll need access to a computer with the following installed:

- `Python` (> 3.6)
- `NumPy` (>= 1.13.3)
- `SciPy` (>= 0.19.1)
- `scikit-learn` (>= 0.19.1)

The easiest way to install all of these together is with Anaconda (Windows, Mac & Linux installers available).

Finally, you'll need the datasets we'll be using - you can download this from https://github.com/jonhare/DISCnetMachineLearningCourse/raw/master/Monday/ml101-tutorial/data.zip.

A data set for experimentation

For the purposes of this tutorial we're going to play with a dataset of internet newsgroup posts. 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.

Start by unzipping the datasets downloaded above and navigating to the `data/twenty_newsgroups` folder. Look at how the data set is structured; there are two folders - one with data for training models, and one for testing how well a model works. Within each of the training and testing folders are 20 folders representing the 20 different newsgroups. Within these folders are the actual messages posted on the newsgroups, with one file per message. Spend some time to open a few of the files in a text editor to see their contents.

Now open a python interpreter (either `python3` or `ipython3`) to get started learning how to use `scikit-learn`.

We're going to start by loading the dataset into memory. `scikit-learn` contains a number of tools that can help us do this. In order to get faster execution times for the initial parts of this tutorial 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 using the `sklearn.datasets.load_files` function as follows (change the path to match where your copy of the data is stored):

```
>>> from sklearn.datasets import load_files
>>> twenty_train = load_files('/path/to/data/twenty_newsgroups/train',
... categories=categories, shuffle=True, random_state=42, encoding='latin1')
```

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.christian']
```

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: clipper@mccarthy.csd.uwo.ca (Khun Yee Fung)
Subject: Re: looking for circle algorithm faster than Bresenhams
Organization: Department of Computer Science, The University of Western

>>> 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, 0, 2, 2, 0, 1, 1, 3, 3, 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
alt.atheism
sci.med
sci.med
alt.atheism
comp.graphics
comp.graphics
soc.religion.christian
soc.religion.christian
sci.med
```

You can notice that the samples have been shuffled randomly (with a fixed random number generator 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.

Building a Basic "Bag of Words" Feature Extractor

In order to perform machine learning on text documents, we first need to turn the text content into numerical featurevectors. 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 might not be manageable on the computer you're using today.

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 stop-words 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` also supports counts of N-grams of words or consecutive characters. N-grams are runs of consecutive characters or words, so for example in the case of word bi-grams, every consecutive pair of words would be a feature. Once fitted, the vectorizer has built a dictionary of feature indices:

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

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. The number of times a term occurs in a document, divided by the number of terms in a document is called the **term frequency** (*tf*).

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. In order to achieve this we can weight terms on the basis of the **inverse document frequency** (*idf*). The *document frequency* is the number of documents a given word occurs in; the inverse document frequency is often defined as the number of documents divided by the *df*.

Combining *tf* and *idf* results in a *family* of weightings (*tf* is usually multiplied by *idf*, but there a few different variations of how *idf* is computed) known as "Term Frequency - Inverse Document Frequency" (tf–idf).

Both **tf** and **tf–idf** can be computed using `scikit-learn` as follows:

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

In the above 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_counts)
>>> X_train_tfidf.shape
(2257, 35788)
```

Rather than transforming the raw counts with the `TfidfTransformer`, it is alternatively possible to use the `TfidfVectorizer` to directly parse the dataset. The advantage of doing this is that it can automatically filter out less informative words on the basis of stop-words, document frequency, etc.:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> tfidf_vect = TfidfVectorizer(stop_words='english',max_df=0.5,min_df=2)
>>> X_train_tfidf = tfidf_vect.fit_transform(twenty_train.data)
>>> X_train_tfidf.shape
(2257, 18188)
```

As you can see from the output, the number of features was reduced from 35788 to 18188 using this approach.

Exercise: Explore different parameters for the `TfidfVectorizer` - take a look at the documentation here to see what else it can do

---

Exploring K-Means clustering using scikit-learn

Now we've extracted features from our training documents we're in a position to experiment with clustering. We'll use K-Means as its one of the most intuitive clustering methods, although it does have a few limitations.

K-Means clustering with 4 clusters can be achieved as follows:

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(4)
>>> km.fit(X_train_tfidf)
```

The assignments of the original posts to cluster id is given by `km.labels_` once `km.fit(..)` has been called. The centroids of the clusters is given by `km.cluster_centers_`. Intuitively, the vector that describes the centre of a cluster is just like any other featurevector, can every element can be interpreted as the number of times a specific term occurs (or the tf-idf weight of a specific term) in a hypothetical document. An interesting way to explore what each cluster is representing is to calculate and print the top weighted (either by occurrence or tf-idf) terms for that cluster:

```
>>> order_centroids = km.cluster_centers_.argsort()[:, ::-1]
>>> terms = tfidf_vect.get_feature_names()
>>> for i in range(4):
...     print("Cluster %d:" % i, end="")
...     for ind in order_centroids[i, :10]:
```

```
...         print(' %s' % terms[ind], end="")
...     print()
...
Cluster 0: keith caltech livesey sgi wpd solntze jon schneider cco morality
Cluster 1: pitt geb banks gordon cs cadre dsl shameful n3jxp surrender
Cluster 2: com university article posting graphics host nntp know msg like
Cluster 3: god jesus people believe bible christians faith hell christian church
```

Notice how when we performed the clustering that we chose to use 4 clusters. This was intentional, as we know that our data comes from 4 different newsgroups. We might hope that the clustering is able to separate out the 4 different newsgroups automatically, although this is in no-way guaranteed as the clustering is purely unsupervised. Your clusters might vary slight from those printed above (in particular with respect to the order of the clusters); this is because KMeans has a random initialisation element to the algorithm.

A number of different *metrics* exist that allow us to measure how well the clusters fit the known distribution of underlying newsgroups. One such metric is the *homogeneity* which is a measure of how pure the clusters are with respect to the known groupings:

```
>>> from sklearn import metrics
>>> print("Homogeneity: %0.3f" % metrics.homogeneity_score(
... twenty_train.target, km.labels_))
Homogeneity: 0.334
```

Homogeneity scores vary between 0 and 1; a score of 1 indicates that the clusters match the original label distribution exactly.

Exercise: Can you print out which cluster each document belongs to? Hint: use `km.labels_` to get the cluster assignment of each document index, and `twenty_train.filenames` to get the filenames of the corresponding documents.

Exercise: Explore what happens if you make the number of clusters larger. What do you notice? Do the clusters of posts to the mailing lists begin to make more intuitive sense?

Building a predictive model using K-Nearest-Neighbours

Now that we have our training features and the known newsgroup label for each post, we can train a classifier to try to predict the category of a post. Let's start with a KNN classifier, which provides a simple baseline, although is perhaps not the best classifier for this task:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> clf = KNeighborsClassifier(n_neighbors=3).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/vectorizers, since they have already been fit to the training set:

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

>>> predicted = clf.predict(X_new_tfidf)

>>> for doc, category in zip(docs_new, predicted):
...     print('%r => %s' % (doc, twenty_train.target_names[category]))
...
'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([('tfidf', TfidfVectorizer()),
...                      ('clf', KNeighborsClassifier(n_neighbors=3))
... ])
```

The names `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.target)
```

Evaluation of the performance on the test set

Evaluating the predictive accuracy (average number of correct predictions divided by the number of total predictions) of the model is equally easy:

```
>>> import numpy as np
>>> twenty_test = load_files('/path/to/data/twenty_newsgroups/test',
... categories=categories, shuffle=True, random_state=42, encoding='latin1')
>>> docs_test = twenty_test.data
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.77230359520639147
```

Thie means we achieved a 77.2% 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 (when used with suitable features). We can change the learner by just plugging a different classifier object into our pipeline:

```
>>> from sklearn.linear_model import SGDClassifier
>>> text_clf = Pipeline([('tfidf', TfidfVectorizer()),
...                       ('clf', SGDClassifier(loss='hinge', penalty='l2',
...                                             alpha=1e-3, max_iter=5, tol=None,
...                                             random_state=42)),
... ])
>>> _ = text_clf.fit(twenty_train.data, twenty_train.target)
>>> predicted = text_clf.predict(docs_test)
>>> np.mean(predicted == twenty_test.target)
0.90812250332889477
```

Note: In the above code we're using a `SGDClassifier`. The `SGDClassifier` is a special type of classifier that performs an optimisation process called Stochastic Gradient Descent to minimise the error or loss function. When we use Hinge Loss (`loss='hinge'`) the objective function is equivalent to a Support Vector Machine. The penalty term (`penalty='l2'`) applies an *l2 regulariser* to the optimiser; this has the effect of constraining the magnitudes of the values being optimised. The strength of this regulariser is controlled by the `alpha` parameter. We'll talk more about Support Vector Machines, regularisation and gradient methods later in the week. You should also know that scikit-learn has many other implementations of linear Support Vector Machines (as well as non-linear or Kernel SVMs), which use different mechanisms for optimising the solution (see http://scikit-learn.org/stable/modules/svm.html#svm). The `SGDClassifier` is a good choice for large-scale problems because it uses less memory than the `LinearSVC` (Linear Support Vector Classifier) class.

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

```
>>> from sklearn import metrics
>>> print(metrics.classification_report(twenty_test.target, predicted,
...     target_names=twenty_test.target_names))
                        precision    recall  f1-score   support

           alt.atheism       0.96      0.79      0.87       319
         comp.graphics       0.87      0.98      0.92       389
               sci.med       0.94      0.89      0.91       396
soc.religion.christian       0.89      0.95      0.92       398

           avg / total       0.91      0.91      0.91      1502

>>> metrics.confusion_matrix(twenty_test.target, predicted)
array([[251,  11,  17,  40],
       [  1, 382,   3,   3],
       [  4,  37, 351,   4],
       [  5,  11,   2, 380]])
```

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` (and `TfidfVectorizer` if you followed the exercise and looked at the documentation). Classifiers tend to have many parameters as well; e.g., `KNeighborsClassifier` includes parameter for the number of neighbours 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. Let's use this to explore whether we can make the `KNeighborsClassifier` perform as well as our linear SVM. We'll try out classifiers on either words or bi-grams, with or without idf, and with a K (number of neighbours) ranging from 1 to 7 (odd numbers only):

```
>>> text_clf = Pipeline([('tfidf', TfidfVectorizer()),
...                      ('clf', KNeighborsClassifier(n_neighbors=3))
... ])
>>> from sklearn.model_selection import GridSearchCV
>>> parameters = {'tfidf__ngram_range': [(1, 1), (1, 2)],
...               'tfidf__use_idf': (True, False),
...               'clf__n_neighbors': (1, 3, 5, 7)
... }
```

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 available and use 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.target[:400])
```

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'])[0]]
'soc.religion.christian'
```

The object's `best_score_` and `best_params_` attributes store the best mean score and the parameters setting corresponding to that score:

```
>>>
>>> gs_clf.best_score_
0.69999999999999996

>>> for param_name in sorted(parameters.keys()):
...     print("%s: %r" % (param_name, gs_clf.best_params_[param_name]))
...
clf__n_neighbors: 5
tfidf__ngram_range: (1, 2)
tfidf__use_idf: True
```

A more detailed summary of the search is available in the `gs_clf.cv_results_` attribute.

Exercise: Try using `GridSearchCV` together with the `SGDClassifier` SVM classifier we defined above to find the optimal `alpha`, together with the optimal ngram range and use idf parameters for the vectoriser. How does the classification accuracy compare to the K-Nearest-Neighbours classifier? How does the performance change if more training data is used for both classifiers?

Exercise: Can you build a classifier for the entire 20 class dataset? What is the performance, and how does it compare to the 4 classes we have been experimenting with?

Exploring different classification problems

Language Identification

> **Exercise:** 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. Download the starter code from https://github.com/jonhare/DISCnetMachineLearningCourse/raw/master/Monday/ml101-tutorial/train_language_model.py & save it as `train_language_model.py`:

To run the code use `python3 train_language_model.py path/to/data/language/paragraphs` (change the path to match where your copy of the data is stored)

Sentiment Analysis

> **Exercise:** 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. Download the starter code from https://github.com/jonhare/DISCnetMachineLearningCourse/raw/master/Monday/ml101-tutorial/sentiment.py & save it as `sentiment.py`:

To run the code use `python3 sentiment.py path/to/data/movie_reviews/txt_sentoken` (change the path to match where your copy of the data is stored)

Build a command-line text classification tool

> **Exercise:** 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 points if the utility is able to give a confidence level for its predictions.

Where to go 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 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`.