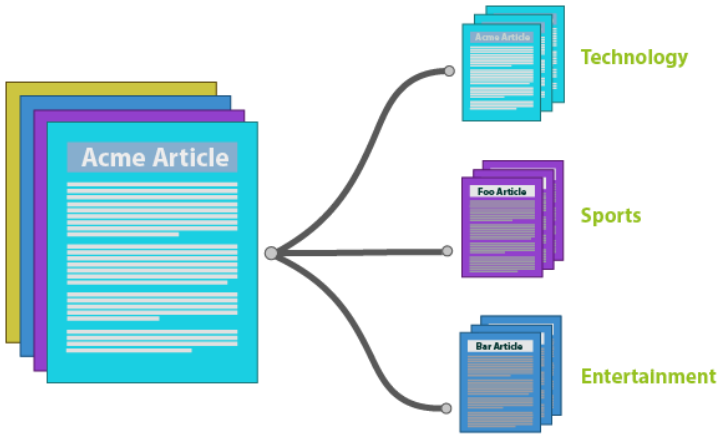
Javed Shaikh Follow
Building better future inspires me. Personal opinions.
Jul 23, 2017 · 7 min read

Machine Learning, NLP: Text Classification using scikit-learn, python and NLTK.



Text Classification

Latest Update:
I have uploaded the complete code (Python and Jupyter notebook) on GitHub: https://github.com/javedsha/text-classification

Document/Text classification is one of the important and typical task in *supervised* machine learning (ML). Assigning categories to documents, which can be a web page, library book, media articles, gallery etc. has many applications like e.g. spam filtering, email routing, sentiment analysis etc. In this article, I would like to demonstrate how we can do text classification using python, scikit-learn and little bit of NLTK.

Disclaimer: *I am new to machine learning and also to blogging (First). So, if there are any mistakes, please do let me know. All feedback appreciated.*

Let's divide the classification problem into below steps:

1. Prerequisite and setting up the environment.

2. Loading the data set in jupyter.
3. Extracting features from text files.
4. Running ML algorithms.
5. Grid Search for parameter tuning.
6. Useful tips and a touch of NLTK.

Step 1: Prerequisite and setting up the environment

The prerequisites to follow this example are python version **2.7.3** and jupyter notebook. **You can just install anaconda and it will get everything for you.** Also, little bit of python and ML basics including text classification is required. We will be using scikit-learn (python) libraries for our example.

Step 2: Loading the data set in jupyter.

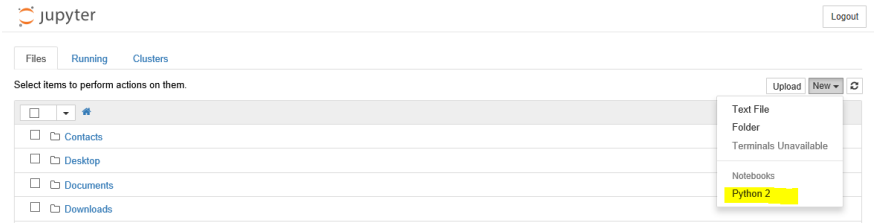
The data set will be using for this example is the famous "20 Newsgoup" data set. About the data from the original website:

> *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 my knowledge, it was originally collected by Ken Lang, probably for his Newsweeder: Learning to filter netnews paper, 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.*

This data set is in-built in scikit, so we don't need to download it explicitly.

i. Open command prompt in windows and type 'jupyter notebook'. This will open the notebook in browser and start a session for you.

ii. Select New > Python 2. You can give a name to the notebook - *Text Classification Demo 1*



iii. Loading the data set: (this might take few minutes, so patience)

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

*Note: Above, we are only loading the **training** data. We will load the test data separately later in the example.*

iv. You can check the target names (categories) and some data files by following commands.

```
twenty_train.target_names #prints all the categories
print("\n".join(twenty_train.data[0].split("\n")[:3]))
#prints first line of the first data file
```

Step 3: Extracting features from text files.

Text files are actually series of words (ordered). In order to run machine learning algorithms we need to convert the text files into numerical feature vectors. We will be using **bag of words** model for our example. Briefly, we segment each text file into words (for English splitting by space), and count # of times each word occurs in each document and finally assign each word an integer id. **Each unique word in our dictionary will correspond to a feature (descriptive feature).**

Scikit-learn has a high level component which will create feature vectors for us 'CountVectorizer'. More about it here.

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

Here by doing '*count_vect.fit_transform(twenty_train.data)*', we are learning the vocabulary dictionary and it returns a Document-Term matrix. [n_samples, n_features].

TF: Just counting the number of words in each document has 1 issue: it will give more weightage to longer documents than shorter documents. To avoid this, we can use frequency (**TF - Term Frequencies**) i.e. #count(word) / #Total words, in each document.

TF-IDF: Finally, we can even reduce the weightage of more common words like (the, is, an etc.) which occurs in all document. This is called as **TF-IDF i.e Term Frequency times inverse document frequency.**

We can achieve both using below line of code:

```
from sklearn.feature_extraction.text import TfidfTransformer
tfidf_transformer = TfidfTransformer()
X_train_tfidf =
tfidf_transformer.fit_transform(X_train_counts)
X_train_tfidf.shape
```

The last line will output the dimension of the Document-Term matrix -> (11314, 130107).

Step 4. Running ML algorithms.

There are various algorithms which can be used for text classification. We will start with the most simplest one 'Naive Bayes (NB)' (*don't think it is too Naive!* 😃)

You can easily build a NBclassifier in scikit using below 2 lines of code: (note - there are many variants of NB, but discussion about them is out of scope)

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

This will train the NB classifier on the training data we provided.

Building a pipeline: We can write less code and do all of the above, by building a pipeline as follows:

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

text_clf = text_clf.fit(twenty_train.data,
twenty_train.target)
```

The names 'vect' , 'tfidf' and 'clf' are arbitrary but will be used later.

Performance of NB Classifier: Now we will test the performance of the NB classifier on **test set**.

```
import numpy as np
twenty_test = fetch_20newsgroups(subset='test',
shuffle=True)
predicted = text_clf.predict(twenty_test.data)
np.mean(predicted == twenty_test.target)
```

The accuracy we get is **~77.38%**, which is not bad for start and for a naive classifier. Also, congrats!!! you have now written successfully a text classification algorithm 👍

Support Vector Machines (SVM): Let's try using a different algorithm SVM, and see if we can get any better performance. More about it here.

```
>>> from sklearn.linear_model import SGDClassifier
```

```
>>> text_clf_svm = Pipeline([('vect', CountVectorizer()),
...                      ('tfidf', TfidfTransformer()),
...                      ('clf-svm',
SGDClassifier(loss='hinge', penalty='l2',
...                                            alpha=1e-3,
n_iter=5, random_state=42)),
... ])


>>> _ = text_clf_svm.fit(twenty_train.data,
twenty_train.target)


>>> predicted_svm = text_clf_svm.predict(twenty_test.data)
>>> np.mean(predicted_svm == twenty_test.target)
```

The accuracy we get is **~82.38%.** Yipee, a little better 👌

Step 5. Grid Search

Almost all the classifiers will have various parameters which can be tuned to obtain optimal performance. Scikit gives an extremely useful tool 'GridSearchCV'.

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

Here, we are creating a list of parameters for which we would like to do performance tuning. All the parameters name start with the classifier name (remember the arbitrary name we gave). E.g. vect__ngram_range; here we are telling to use unigram and bigrams and choose the one which is optimal.

Next, we create an instance of the grid search by passing the classifier, parameters and n_jobs=-1 which tells to use multiple cores from user machine.

```
gs_clf = GridSearchCV(text_clf, parameters, n_jobs=-1)
gs_clf = gs_clf.fit(twenty_train.data, twenty_train.target)
```

This might take few minutes to run depending on the machine configuration.

Lastly, to see the best mean score and the params, run the following code:

```
gs_clf.best_score_
gs_clf.best_params_
```

The accuracy has now increased to **~90.6% for the NB classifier** (not so naive anymore! 😄) and the corresponding parameters are {'clf__alpha': 0.01, 'tfidf__use_idf': True, 'vect__ngram_range': (1, 2)}.

Similarly, we get improved accuracy **~89.79%** for SVM classifier with below code. *Note: You can further optimize the SVM classifier by tuning other parameters. This is left up to you to explore more.*

```
>>> from sklearn.model_selection import GridSearchCV
>>> parameters_svm = {'vect__ngram_range': [(1, 1), (1, 2)],
...               'tfidf__use_idf': (True, False),
...               'clf-svm__alpha': (1e-2, 1e-3),
... }
gs_clf_svm = GridSearchCV(text_clf_svm, parameters_svm,
n_jobs=-1)
gs_clf_svm = gs_clf_svm.fit(twenty_train.data,
twenty_train.target)
gs_clf_svm.best_score_
gs_clf_svm.best_params_
```

Step 6: Useful tips and a touch of NLTK.

1. **Removing stop words:** (the, then etc) from the data. You should do this only when stop words are not useful for the underlying problem. In most of the text classification problems, this is indeed not useful. Let's see if removing stop words increases the accuracy. Update the code for creating object of CountVectorizer as follows:

```
>>> from sklearn.pipeline import Pipeline
>>> text_clf = Pipeline([('vect',
CountVectorizer(stop_words='english')),
```

```
...                      ('tfidf', TfidfTransformer()),
...                      ('clf', MultinomialNB()),
... ])
```

This is the pipeline we build for NB classifier. Run the remaining steps like before. This improves the accuracy from **77.38% to 81.69%** (that is too good). *You can try the same for SVM and also while doing grid search.*

2. **FitPrior=False:** When set to false for MultinomialNB, a uniform prior will be used. This doesn't helps that much, but increases the accuracy from 81.69% to 82.14% (not much gain). *Try and see if this works for your data set.*

3. Stemming: From Wikipedia, stemming is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form. E.g. A stemming algorithm reduces the words "fishing", "fished", and "fisher" to the root word, "fish".

We need NLTK which can be installed from here. NLTK comes with various stemmers (*details on how stemmers work are out of scope for this article*) which can help reducing the words to their root form. Again use this, if it make sense for your problem.

Below I have used Snowball stemmer which works very well for English language.

```
import nltk
nltk.download()


from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer("english", ignore_stopwords=True)


class StemmedCountVectorizer(CountVectorizer):
    def build_analyzer(self):
        analyzer = super(StemmedCountVectorizer,
self).build_analyzer()
        return lambda doc: ([stemmer.stem(w) for w in
analyzer(doc)])


stemmed_count_vect =
StemmedCountVectorizer(stop_words='english')
```

```
text_mnb_stemmed = Pipeline([('vect', stemmed_count_vect),
...                      ('tfidf', TfidfTransformer()),
...                      ('mnb',
MultinomialNB(fit_prior=False)),
... ])


text_mnb_stemmed = text_mnb_stemmed.fit(twenty_train.data,
twenty_train.target)


predicted_mnb_stemmed =
text_mnb_stemmed.predict(twenty_test.data)


np.mean(predicted_mnb_stemmed == twenty_test.target)
```

The accuracy with stemming we get is ~81.67%. Marginal improvement in our case with NB classifier. You can also try out with SVM and other algorithms.

Conclusion: We have learned the classic problem in NLP, text classification. We learned about important concepts like bag of words, TF-IDF and 2 important algorithms NB and SVM. We saw that for our data set, both the algorithms were almost equally matched when optimized. *Sometimes*, if we have enough data set, choice of algorithm can make hardly any difference. We also saw, how to perform grid search for performance tuning and used NLTK stemming approach. You can use this code on your data set and see which algorithms works best for you.

Update: If anyone tries a different algorithm, please share the results in the comment section, it will be useful for everyone.

Please let me know if there were any mistakes and feedback is welcome ✌

Recommend, comment, share if you liked this article.

References:

http://scikit-learn.org/ (code)

http://qwone.com/~jason/20Newsgroups/ (data set)