DocumentClassificationWithSVM

November 3, 2021

[43]:
```
'''
 The data set that we'll use for this problem is available with the␣
 ↪scikit-learn library.
 Scikit-learn contains a number of data sets that can be used to train and␣
 ↪validate models.
 We'll use the fetch 20 newsgroups module to retrieve the data set.
 This contains roughly 20, 000 newsgroup documents which are split across 20␣
 ↪newsgroups.

 Each one of these 20 newsgroups corresponds to a particular topic.
 The return value from this function is a dictionary and these are the keys␣
 ↪within the dictionary.
'''
from sklearn.datasets import fetch_20newsgroups
tewnty_train=fetch_20newsgroups(subset='train',shuffle=True)
```

http://qwone.com/~jason/20Newsgroups/

[44]:
```
'''
The return value from this function is a dictionary and these are the keys␣
 ↪within the dictionary.

The data key is what contains our training data. These are our newsgroup␣
 ↪documents.

Target names are the newsgroups to which these documents belong.
These are the labels or the y values associated with each document.
'''
tewnty_train.keys()
```

[44]:
```
dict_keys(['data', 'filenames', 'target_names', 'target', 'DESCR'])
```

[45]:
```
'''
Here is a document sample from the training data set.
This is an email related to cars.

'''
```

```
print(tewnty_train.data[0])
```

```
From: lerxst@wam.umd.edu (where's my thing)
Subject: WHAT car is this!?
Nntp-Posting-Host: rac3.wam.umd.edu
Organization: University of Maryland, College Park
Lines: 15

 I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In addition,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.

Thanks,
- IL
   ---- brought to you by your neighborhood Lerxst ----
```

[46]:
```
'''
Here are the 20 newsgroups to which these documents belong.
They range from sports to computers to politics.
'''
tewnty_train.target_names
```

[46]:
```
['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
```

```
 'talk.politics.guns',
 'talk.politics.mideast',
 'talk.politics.misc',
 'talk.religion.misc']
```

[47]:
```
'''
Our categorical variables need to be expressed in numeric form and that's what␣
 ↪the target key holds.
'''
tewnty_train.target
```

[47]:
```
array([7, 4, 4, …, 3, 1, 8])
```

[48]:
```
'''
Every document is x data and and we will represent it in numeric form using the␣
 ↪count vectorizer.

We'll call the count vectorizers fit transform method on our training data.

The output of the count vectorizer is a sparse matrix.

Every word is identified uniquely using its document ID and its unique word ID
and the frequency of the word in that document is specified.

Here is the shape of our sparse matrix.

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

[48]:
```
(11314, 130107)
```

[49]:
```
'''
You can explore the output of the count vectorizer on the very first document.
Here you can see the document ID, the word ID and the associated frequency.
'''
print(X_train_counts[0])
```

```
  (0, 56979)    3
  (0, 75358)    2
  (0, 123162)   2
  (0, 118280)   2
  (0, 50527)    2
  (0, 124031)   2
  (0, 85354)    1
```

```
  (0, 114688)   1
  (0, 111322)   1
  (0, 123984)   1
  (0, 37780)    5
  (0, 68532)    3
  (0, 114731)   5
  (0, 87620)    1
  (0, 95162)    1
  (0, 64095)    1
  (0, 98949)    1
  (0, 90379)    1
  (0, 118983)   1
  (0, 89362)    3
  (0, 79666)    1
  (0, 40998)    1
  (0, 92081)    1
  (0, 76032)    1
  (0, 4605)     1
  :     :
  (0, 37565)    1
  (0, 113986)   1
  (0, 83256)    1
  (0, 86001)    1
  (0, 51730)    1
  (0, 109271)   1
  (0, 128026)   1
  (0, 96144)    1
  (0, 78784)    1
  (0, 63363)    1
  (0, 90252)    1
  (0, 123989)   1
  (0, 67156)    1
  (0, 128402)   2
  (0, 62221)    1
  (0, 57308)    1
  (0, 76722)    1
  (0, 94362)    1
  (0, 78955)    1
  (0, 114428)   1
  (0, 66098)    1
  (0, 35187)    1
  (0, 35983)    1
  (0, 128420)   1
  (0, 86580)    1
```

[50]:
```
count_vect.vocabulary_.get("where")
```

[50]: 124031

[51]:
```
'''
 I'm going to parse this output of the count vectorizer through a␣
 ↪TfidfTransformer.

 The TfidfTransformer is different from the Tfidf vectorizer in one significant␣
 ↪way.

 The Tfidf vectorizer worked directly on documents and produced a bag of words␣
 ↪with corresponding Tfidf course.

 The TfidfTransformer on the other hand requires a bag of words as its input.

 That's why the output of the count vectorizer be passed into the␣
 ↪TfidfTransformer.

 The count vectorizer plus the TfidfTransformer is equal to the tfidfvectorizer.
 ↪
'''

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

[51]: (11314, 130107)

[52]:
```
'''
Let's print this course for one document.
Here you can see a mapping of document ID word ID and the corresponding tfidf␣
 ↪score.
'''
print(X_train_tfidf[0])
```

```
  (0, 128420)   0.04278499079283093
  (0, 128402)   0.05922294083277842
  (0, 128026)   0.060622095889758885
  (0, 124931)   0.08882569909852546
  (0, 124031)   0.10798795154169122
  (0, 123989)   0.08207027465330353
  (0, 123984)   0.036854292634593756
  (0, 123796)   0.049437556160455476
  (0, 123292)   0.14534718515938805
  (0, 123162)   0.2597090245735688
  (0, 118983)   0.037085978050619146
  (0, 118280)   0.2118680720828169
```

```
  (0, 115475)   0.042472629883573
  (0, 114731)   0.14447275512784058
  (0, 114688)   0.06214070986309586
  (0, 114579)   0.03671830826216751
  (0, 114455)   0.12287762616208957
  (0, 114428)   0.05511105154696676
  (0, 113986)   0.17691750674853082
  (0, 111322)   0.01915671802495043
  (0, 109581)   0.10809248404447917
  (0, 109271)   0.10844724822064673
  (0, 108252)   0.07526015712540636
  (0, 106116)   0.09869734624201922
  (0, 104813)   0.08462829788929047
  :     :
  (0, 56979)    0.057470154074851294
  (0, 51793)    0.13412921037839678
  (0, 51730)    0.09714744057976722
  (0, 50527)    0.054614286588587246
  (0, 50111)    0.08452530453354308
  (0, 48620)    0.11603642565244157
  (0, 48618)    0.10015015488700592
  (0, 45295)    0.06621689096551565
  (0, 42876)    0.04951998622750595
  (0, 40998)    0.0780136819691811
  (0, 37780)    0.38133891259493113
  (0, 37565)    0.03431760442478462
  (0, 37433)    0.06908779999621749
  (0, 35983)    0.03770448563619875
  (0, 35612)    0.1328075333935896
  (0, 35187)    0.09353930598317124
  (0, 34995)    0.16713176431412632
  (0, 34181)    0.08716420445779295
  (0, 32311)    0.029911858621703466
  (0, 28615)    0.10278592160069082
  (0, 27436)    0.037098931990947055
  (0, 26073)    0.09534869974107982
  (0, 18299)    0.138749083899155
  (0, 16574)    0.14155752531572685
  (0, 4605)     0.06332603952480323
```

[53]:
```
'''
Once we've set up our data correctly, using a support vector machine estimator␣
 ↪is very easy.
We instantiate the linear SVC and pass in a number of input arguments.

The penalty function that we are going to use is the L2 norm penalty.
```

```
Another criteria that we can specify is the tolerance for stopping training on␣
 ↪our model.

If the losses that we calculate on the objective function go below this␣
 ↪tolerance value,
we'll assume that our model is good enough and stop training.

Notice that in order to prepare the data set, we performed a series of␣
 ↪transformations on it.
We first used the count vectorizer and then the TfidfTransformer and then pass␣
 ↪that
output to our linear SVC estimator.

'''
from sklearn.svm import LinearSVC
clf_svc=LinearSVC(penalty='l2',dual=False,tol=1e-3)
clf_svc.fit(X_train_tfidf,tewnty_train.target)
```

[53]:
```
LinearSVC(dual=False, tol=0.001)
```

[54]:
```
'''
Scikit-learn provides a very handy tool called the pipeline which allows a␣
 ↪linear sequence
of data transformations to be chain.

Notice that we instantiate a pipeline and within the pipeline we specified the␣
 ↪series of
transformations that we want performed.

We can now simply execute this pipeline by calling the fit method and our␣
 ↪training data will
be passed through each of these steps.

We don't need to call these steps individually.
'''

from sklearn.pipeline import Pipeline
clf_svc_pipeline=Pipeline([('vect',CountVectorizer()),
                           ('tfidf',TfidfTransformer()),
                           ('clf',LinearSVC(penalty="l2",dual=False,tol=0.001))])
clf_svc_pipeline.fit(tewnty_train.data,tewnty_train.target)
```

[54]:
```
Pipeline(steps=[('vect', CountVectorizer()), ('tfidf', TfidfTransformer()),
                ('clf', LinearSVC(dual=False, tol=0.001))])
```

[55]:
```
'''
Let's go ahead and get the test data and we can pass this test data through the␣
 ↪same pipeline.
'''
tewnty_test=fetch_20newsgroups(subset='test',shuffle=True)
```

[56]:
```
'''
The last step of the pipeline we'll use our linear support vector classifier␣
 ↪for prediction.
This is the same classifier that we just trained.
'''
predicted=clf_svc_pipeline.predict(tewnty_test.data)
```

[57]:
```
'''
One way to measure how well our classifier performs is to calculate the␣
 ↪accuracy of our predictions.
Let's see how many of our predicted labels are equal to the actual labels.
Our support vector machine classifier performs pretty well. It has an accuracy␣
 ↪of 85%.

'''
from sklearn.metrics import accuracy_score
acc_svm=accuracy_score(tewnty_test.target,predicted)
acc_svm
```

[57]: 0.8532926181625067

[58]:
```
'''
 Let's tweak our model a little bit. Instead of using the L2 norm as our␣
 ↪penalty function,
 we will use the L1 norm. Go ahead and hit shift + enter to execute all the␣
 ↪cells.
 You'll find that the accuracy of this model is around 81. 5%. It has fallen a␣
 ↪bit.
'''
clf_svc_pipeline=Pipeline([('vect',CountVectorizer()),
                           ('tfidf',TfidfTransformer()),
                           ('clf',LinearSVC(penalty="l1",dual=False,tol=0.001))])
clf_svc_pipeline.fit(tewnty_train.data,tewnty_train.target)
tewnty_test=fetch_20newsgroups(subset='test',shuffle=True)
predicted=clf_svc_pipeline.predict(tewnty_test.data)
acc_svm=accuracy_score(tewnty_test.target,predicted)
acc_svm
```

[58]: 0.8150557620817844

[60]:
```
'''
Let's make a little change to our pipeline.
Instead of using the output of the TfidfTransformer, we'll use the output of␣
 ↪the count vectorizer directly
to feed into our support vector classifier.

Train the pipeline and perform predictions on the test data.
You can see that the accuracy is around 79. 8%. Compare this with the accuracy␣
 ↪of 85%
when we use the exact same model but the output of the Tfidfvectorizer.

This gives us an idea of the impact of the Tfidf scores in our classification␣
 ↪model.
'''
clf_svc_pipeline=Pipeline([('vect',CountVectorizer()),
                           ('clf',LinearSVC(penalty="l2",dual=False,tol=0.001))])
clf_svc_pipeline.fit(tewnty_train.data,tewnty_train.target)
tewnty_test=fetch_20newsgroups(subset='test',shuffle=True)
predicted=clf_svc_pipeline.predict(tewnty_test.data)
acc_svm=accuracy_score(tewnty_test.target,predicted)
acc_svm
```

[60]:
```
0.7963356346255974
```

[ ]:
```

```