IN4080_2019_Mandatory_2_A

September 25, 2019

1 IN4080 2019, Mandatory assignment 2, part A

1.0.1 About the assignment

Your answer should be delivered in devilry no later than Friday, 11 October at 23:59

Mandatory assignment 2 consists of two parts

- Part A on text classification (=this file)
- Part B on tagging and sequence classification (separate document)

You should answer both parts. It is possible to get 50 points on each part, 100 points altogether. You are required to get at least 60 points to pass. It is more important that you try to answer each question than that you get it correct.

1.0.2 General requirements:

- We assume that you have read and are familiar with IFI's requirements and guidelines for mandatory assignments
  - https://www.uio.no/english/studies/examinations/compulsory-activities/mn-ifi-mandatory.html
  - https://www.uio.no/english/studies/examinations/compulsory-activities/mn-ifi-guidelines.html
- This is an individual assignment. You should not deliver joint submissions.
- You may redeliver in Devilry before the deadline, but include all files in the last delivery. Only the last delivery will be read!
- If you deliver more than one file, put them into a zip-archive.
- Name your submission your_username_in4080_mandatory_2

The delivery can take one of two forms:

- Alternative I:
  - Deliver the code files.
  - In addition, deliver a separate pdf file containing results from the runs together with answers to the text questions.
- Alternative II:
  - A jupyter notebook containing code, answers to the text questions in markup and optionally results from the runs.

  - In addition, a pdf-version of the notebook where (in addition) all the results of the runs are included.

Whether you use the first or second alternative, make sure that the code runs at the IFI machines after

- export PATH=$PATH:/opt/ifi/python-3.7/bin/

1.0.3 Goal of part A

In this part you will get experience with

- setting up and running experiments
- splitting your data in development and test data
- n-fold cross-validation
- evaluation and evaluation measures
- models for text classification
- Naive Bayes vs Logistic Regression
- the scikit-learner toolkit
- vectorization of categorical data

As background for the current assignment you should work through two tutorials

- Document classification from the NLTK book, Ch. 6. See (weekly) exercise set 3
- The scikit-learn tutorial on text classification. See exercise set 4.

If you have any questions regarding these two tutorials, we will be happy to answer them during the group/lab sessions.

1.1 Ex 1 First classifier and n-fold cross-validation (10 points)

1.1.1 Part a. Inititial classifier

We will work interactively in python/ipython/notebook. Start by importing the tools we will be using:

[ ]:
```
import nltk
import random
import numpy as np
import scipy as sp
import sklearn
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction import DictVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import BernoulliNB
from sklearn.linear_model import LogisticRegression
```

As data we will use the Movie Reviews Corpus that comes with NLTK.

[ ]:
```
from nltk.corpus import movie_reviews
```

We can import the documents similarly to how it is done in the NLTK book for the Bernoulli Naive Bayes, with one change. We there use the tokenized texts with the command

- `movie_reviews.words(fileid)`

Following the recipe from the scikit "Working with text data" page, we can instead use the raw documents which we can get from NLTK by

- `movie_reviews.raw(fileid)`

scikit will then tokenize for us as part of *count_vect.fit* (or *count_vect.fit_transform*).

[ ]:
```
raw_movie_docs = [(movie_reviews.raw(fileid), category) for
                  category in movie_reviews.categories() for fileid in
                  movie_reviews.fileids(category)]
```

We will shuffle the data and split it into 200 documents for final testing (which we will not use for a while) and 1800 documents for development. Use your birth date as random seed.

[ ]:
```
random.seed(2920)
random.shuffle(raw_movie_docs)
movie_test = raw_movie_docs[:200]
movie_dev  = raw_movie_docs[200:]
```

Then split the development data into 1600 documents for training and 200 for development test set, call them *train_data* and *dev_test_data*. The *train_data* should now be a list of 1600 items, where each is a pair of a text represented as a string and a label. You should then split this *train_data* into two lists, each of 1600 elements, the first, *train_texts*, containing the texts (as strings) for each document, and the *train_target*, containing the corresponding 1600 labels. Do similarly to the *dev_test_data*.

[ ]:
```
"""To be filled in"""
```

It is then time to extract features from the text. We import

[ ]:
```
from sklearn.feature_extraction.text import CountVectorizer
```

We then make a CountVectorizer *v*. This first considers the whole set of training data, to determine which features to extract:

[ ]:
```
v = CountVectorizer()
v.fit(train_texts)
```

Then we use this vectorizer to extract features from the training data and the test data

[ ]:
```
train_vectors = v.transform(train_texts)
dev_test_vectors = v.transform(dev_test_texts)
```

To understand what is going on, you may inspect the *train_vectors* a little more.

We are now ready to train a classifier

[ ]:
```
clf = MultinomialNB()
clf.fit(train_vectors, train_target)
```

We can proceed and see how the classifier will classify one test object, e.g.

```
dev_test_texts[14]
clf.predict(dev_test_vectors[14])
```

We can use the procedure to predict the results for all the test_data, by

```
clf.predict(dev_test_vectors)
```

We can use this for further evaluation (accuracy, recall, precision, etc.) by comparing to *dev_test_targets*. Alternatively, we can get the accuracy directly by

[ ]:
```
clf.score(dev_test_vectors, dev_test_target)
```

Congratulations! You have now made and tested a multinomial naive Bayes text classifier.

1.1.2 Part b.

To make it easier to rerun the experiment and proceed to cross-validation, put most of exercise 1 into a procedure

[ ]:
```
def multi_nb_exp(train_data, test_data):
    """train-data is a list of pairs, where
      first element is representing a text
      second element a string representing a label test-data has the same form

    Train a multinomialNB on train_data, test on test_data
    and return the results
    """
```

Rerun the experiment from part (a) with this procedure and check that the accuracy is the same. Beware, the input should be

```
multi_nb_exp(train_data, dev_test_data)
```

[ ]:
```
%%time
(acc, pred) = multi_nb_exp(train_data, dev_test_data)
print(acc)
```

1.1.3 Part c.

Make a procedure for n-fold cross-validation

[ ]:
```
# We include an option for key word arguments that are passed on to the␣
 ↪experiment.

def n_fold_kwargs(experiment, dev_data, folds=10, **kwargs):
    """
```

```
    experiment is an experiment like multi_nb_exp
    dev_data is a set of pairs
      first element is representing a text
      second element a string representing a label
    **kwargs are passed on to experiment

    Run an n-fold cross-validation of experiment on
    dev_data. return the results.
    """
```

Then run

```
n_fold(multi_nb_exp, dev_data, n=9)
```

and calculate the accuracies for each of the 9 runs, the mean accuracy and the standard deviation of the accuracies. (In case you wonder, the reason why we are running 9-fold instead of, say 10-fold, is simply to get nice round numbers with 1800 items in our development set.)

Deliveries: Code and results of running the code as described. Answers to the questions in part C.

From the large variation in accuracy between the various splits, we recognize the need of using cross-validation (when we have sufficicient time):

1.2 Ex 2 Parameters of the vectorizer (5 points)

We have so far considered the standard parameters for the procedures from scikit-learn. These procedures have, however, many parameters. To get optimal results, we should adjust the parameters. We can go back to the split in exercise 1, and use *train_data* for training various models and *dev_test_data* for testing and comparing them.

To see the parameters for CountVectorizer we may use

```
help(CountVectorizer)
```

In ipython we may alternatively use

```
CountVectorizer?
```

We observe that *CountVectorizer* case folds by default. For a different corpus, it could be interesting to check the effect of this feature, but even the *movie_reviews.raw()* is already in lower case, so that does not have an effect here (You may check!) We could also have explored the effect of excahnging the default tokenizer included in CountVectorizer with the tokenized Brown corpus.

Another interesting feature is *binary*. Setting this to *True* implies only counting whether a word occurs in a document and not how many times it occurs. It could be interesting to see the effect of this feature.

The feature *ngram_range=[1,1]* means we use tokens (=unigrams) only, [2,2] means that we using bigrams only, while [1,2] means both unigrams and bigrams, and so on.

Run experiments where you let *binary* vary over [False, True] and *ngram_range* vary over [[1,1], [1,2], [1,3]]. Run the experiment with 9-fold cross-validation and report the accuracy with the 6

different settings.

Which settings yield the best results?

Deliveries: Code and results of running the code as described. Answers to the questions.

1.3 Ex 3 Logistic Regression (5 points)

We know that Logistic Regression may produce better results than Naive Bayes. We proceed with the same multinomial model for text classification (i.e. we process the data the same way and use the same vectorizer), but exchange the learner with sciki-learn's LogisticRegression. Since logistic regression is slow to train, we restrict ourselves somewhat with respect to which experiments to run. We assume that the best settings above with respect to binary and n-grams will remain the same with logistic regression as with naive Bayes. Therefore, use these best settings and run 9-fold cross-validation. What is the average accuracy?

Deliveries: Code and results of running the code as described. Answer to the question.

1.4 Ex 4 The Bernoulli model (10 points)

We will explore how the Bernoulli naive Bayes, which is used in the NLTK book for the movie data set, can be implemented in scikit-learn. We can follow the NLTK-book and extract features similarly, i.e., using

```
document_features(document)
```

to extract features from *document* into a dictionary. Remember that we here have to use

```
movie_reviews.words(fileid)
```

where we in exercise 1 used

```
movie_reviews.raw(fileid)
```

We can use the same splits as in exercise 1. The train_data will now be a list of 1600 items, where each is a pair where the first element is a dictionary. It remains to transform these dictionaries to numpy arrays of the form scikit accepts. For this, we can use scikit's *DictVectorizer*, see section 4.2.1 in http://scikit-learn.org/stable/modules/feature_extraction.html. (Alternatively, you can extract the features directly as arrays without using dictionaries. It is a little more work, but the experiments will run faster.)

1.4.1 Part a

Make a procedure for the experiment

[ ]:
```
def bernoulli_exp(train_data, test_data, feature_numbs=2000):
    """train-data is a list of pairs, where
      first element is a feature dictionary
      second element a string representing a label
    test-data has the same form

    Train a BernoulliNB on train_data, test on test_data
```

```
    and return the result
    """
```

1.4.2 Part b

Combining this with exercise 2, run

`n_fold(bernoulli_nb_exp, dev_data, n=9)`

and report the results.

Deliveries: Code and results of running the code as described.

1.5 Ex 5 Logistic regression (10 points)

1.5.1 Part a.

Similarly to the Multinomial model, we can also for the Bernoulli model, combine the basic model (i.e. features and vectorization) with a logistic regression learner instead of the naive Bayes learner. Do this, and run the 9-fold cross-validation experiment.

1.5.2 Part b.

The default from NLTK is to use the 2000 most frequent words. We will explore the effect of the size of the feature set. Repeat the 9-fold cross-validation experiment with the 1000, 2000, 5000, 10000 and 20000 most frequent words as features, both with BernoulliNB and with LogisticRegression.

Warning: Running this experiment may take 30-60 min.

Deliveries: Code. A 5x2 table showing the mean accuracies for 9-fold cross-validation for 1000, 2000, 5000, 1000, and 20000 features for the two different classifiers.

1.6 Exercise 6 (10 points)

From the different classifiers with which you have experimented in this exercise set, choose the one with the best performance on the development data. Find the 200 item test set we tucked away in exercise 1. Run the best classifier on this final test set.

Calculate accuracy, recall, precision and F-score for both classes.

Concratulations. You have now completed a full series of experiments on text classification where you tested out various alternatives systematically on the development data before choosing the settings which gave the best results on the development set and finally testing that set-up with the whole development set as training data and the final test set.

Deliveries: Which classifier did you choose? The numbers asked for. Code.

2 THE END of Part A. Proceed to part B