in4080_2020_mandatory_1_b

September 18, 2020

1 IN4080 2020, Mandatory assignment 1, part B

1.0.1 About the assignment

Your answer should be delivered in devilry no later than Friday, 18 September at 23:59

This is the second part of mandatory assignment 1. See part A for general requirements. You are supposed to answer both parts.

1.0.2 Goal of part B

In this part you will get experience with

- setting up and running experiments
- splitting your data into development and test data
- *n*-fold cross-validation
- 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 exercise 3 below for a correction to the NLTK book.
- The scikit-learn tutorial on text classification, http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html all the way up to and including "Evaluation of the performance of the test set".

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 vectorization (10 points)

1.1.1 1a) Initial classifier

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

```
In [1]: 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.

```
In [2]: 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*).

```
In [3]: try:
            raw_movie_docs = [(movie_reviews.raw(fileid), category) for
                              category in movie_reviews.categories() for fileid in
                              movie_reviews.fileids(category)]
        except LookupError:
            nltk.download('movie_reviews')
            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.

```
In [4]: random.seed(2405)
        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*.

```
In [5]: train_data, dev_test_data = movie_dev[200:], movie_dev[:200]
        len(train_data), len(dev_test_data)

Out[5]: (1600, 200)

In [6]: train_texts, train_target = map(list, zip(*train_data))
        dev_test_texts, dev_test_target = map(list, zip(*dev_test_data))
```

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

```
In [7]: 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:

```
In [8]: v = CountVectorizer()
        v.fit(train_texts)

Out[8]: CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
                lowercase=True, max_df=1.0, max_features=None, min_df=1,
                ngram_range=(1, 1), preprocessor=None, stop_words=None,
                strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, vocabulary=None)
```

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

```
In [9]: 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

```
In [10]: clf = MultinomialNB()
         clf.fit(train_vectors, train_target)

Out[10]: MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

We can proceed and see how the classifier will classify one test document, 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

```
In [11]: clf.score(dev_test_vectors, dev_test_target)

Out[11]: 0.81
```

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

1.1.2 1b) Parameters of the vectorizer

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 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/Jupyter notebook 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 exchanging the default tokenizer included in CountVectorizer with other tokenizers.

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.

(Observe, by the way, that this is not the same as the Bernoulli model for text classfication. The Bernoulli model takes into consideration both the probability of being present for the present words, as well as the probability of not being present for the absent words. The binary multinomial model only considers the present words.)

The feature *ngram_range=[1,1]* means we use tokens (=unigrams) only, [2,2] means 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]] and report the accuracy with the 6 different settings in a 2x3 table.

Which settings yield the best results?

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

```
In [12]: arr = np.zeros((2, 3))
         for p in False, True:
             for i in range(1, 4):
                 v = CountVectorizer(binary=p, ngram_range=[1, i])
                 v.fit(train_texts)
                 train_vectors = v.transform(train_texts)
                 dev_test_vectors = v.transform(dev_test_texts)

                 clf = MultinomialNB()
                 clf.fit(train_vectors, train_target)

                 arr[int(p), i-1] = clf.score(dev_test_vectors, dev_test_target)

         arr

Out[12]: array([[0.81 , 0.815, 0.81 ],
                [0.86 , 0.83 , 0.86 ]])

In [13]: import pandas as pd
         pd.DataFrame(data=arr, index=[False, True], columns=["[1, 1]", "[1, 2]", "[1, 3]"])
```

```
Out[13]:        [1, 1]  [1, 2]  [1, 3]
         False    0.81   0.815    0.81
         True     0.86   0.830    0.86
```

1.2 Ex 2 *n*-fold cross-validation (12 points)

1.2.1 2a)

Our *dev_test_data* contain only 200 items. That is a small number for a test set for a binary classifier. The numbers we report may depend to a large degree on the split between training and test data. To get more reliable numbers, we may use *n*-gram cross-validation. We can use the whole *dev_test_data* of 1800 items for this. To get round numbers, we decide to use 9-fold cross-validation, which will put 200 items in each test set.

Use the best settings from exercise 1 and run a 9-fold cross-validation. Report the accuracy for each run, together with the mean and standard deviation of the 9 runs.

In this exercise, you are requested to implement the routine for cross-validation yourself, and not aplly the scikit-learn function.

Deliveries: Code and results of running the code as described. Accuracy for each run, together with the mean and standard deviation of the accuracies for the 9 runs.

```
In [14]: len(train_texts)
```

```
Out[14]: 1600
```

```
In [15]: class KFold:
             def __init__(self, k: int):
                 self._k = k

             def split(self, features: np.ndarray) -> np.ndarray:
                 size = features.shape[0]
                 indices = np.arange(size)
                 test_size = size//self._k

                 for i in range(self._k):
                     test_indices = indices[i*test_size:(1+i)*test_size]
                     # train_indices = indices[np.logical_not(test_indices)]
                     #  print(indices[:i*test_size])
                     #  print(indices[(i+1)*test_size:])
                     train_indices = np.concatenate((indices[:i*test_size], indices[(i+1)*test_
                     yield train_indices, test_indices

         kf = KFold(k=9)
```

```
In [16]: # make indexing easier
         kf_features, kf_targets = np.array(train_texts + dev_test_texts), np.array(train_targ
         kf_features.shape
```

```
Out[16]: (1800,)
```

```
In [17]: accs = np.zeros(9)
         for i, (train_index, test_index) in enumerate(kf.split(kf_features)):
             X_train_text, X_test_text = kf_features[train_index], kf_features[test_index]
             Y_train, Y_test = kf_targets[train_index], kf_targets[test_index]

             v = CountVectorizer(binary=True, ngram_range=(1,3))

             v.fit(X_train_text)
             X_train, X_test = v.transform(X_train_text), v.transform(X_test_text)

             nb = MultinomialNB()

             nb.fit(X_train, Y_train)
             acc = nb.score(X_test, Y_test)
             print("Phase", i, acc)
             accs[i] = acc
         print("Mean:", accs.mean())
         print("stdev:", accs.std())
```

```
Phase 0 0.885
Phase 1 0.875
Phase 2 0.835
Phase 3 0.89
Phase 4 0.87
Phase 5 0.855
Phase 6 0.87
Phase 7 0.78
Phase 8 0.86
Mean: 0.8577777777777779
stdev: 0.03154459903684086
```

1.2.2 2b)

The large variation we see between the results, raises a question regarding whether the optimal settings we found in exercise 1, would also be optimal for another split between training and test.

To find out, we combine the 9-fold cross-validation with the various setting for CountVectorizer. For each of the 6 settings, run 9-fold cross-validation and calculate the mean accuracy. Report the results in a 2x3 table. Answer: Do you see the same as when you only used one test set?

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

```
In [18]: arr = np.zeros((2, 3))
         k = 9
         kf = KFold(k)
         for p in False, True:
             for i in range(1, 4):
                 print("Testing, binary =", p, ", ngram_range =", [1, i])
```

```
                for train_index, test_index in kf.split(kf_features):
                    X_train_text, X_test_text = kf_features[train_index], kf_features[test_in
                    Y_train, Y_test = kf_targets[train_index], kf_targets[test_index]

                    v = CountVectorizer(binary=p, ngram_range=[1, i])

                    v.fit(X_train_text)
                    X_train, X_test = v.transform(X_train_text), v.transform(X_test_text)

                    nb = MultinomialNB()
                    nb.fit(X_train, Y_train)
                    acc = nb.score(X_test, Y_test)
                    arr[int(p), i-1] += acc
        arr /= k
        arr

Testing, binary = False , ngram_range = [1, 1]
Testing, binary = False , ngram_range = [1, 2]
Testing, binary = False , ngram_range = [1, 3]
Testing, binary = True , ngram_range = [1, 1]
Testing, binary = True , ngram_range = [1, 2]
Testing, binary = True , ngram_range = [1, 3]


Out[18]: array([[0.80611111, 0.82722222, 0.82277778],
                [0.82611111, 0.85333333, 0.85777778]])
```

1.3 Ex 3 Logistic Regression (8 points)

We know that Logistic Regression may produce better results than Naive Bayes. We will see what happens if we use Logistic Regression instead of Naive Bayes. We start with the same multinomial model for text classification as in exercises (1) and (2) above (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 consider two settings for the CountVectorizer, the default setting and the setting which gave the best result with naive Bayes (though, this does not have to be the best setting for the logistic regression). For each of the two settings, run 9-fold cross-validation and calculate the mean accuracy. Compare the results in a 2x2 table where one axis is Naive Bayes vs. Logistic Regression and the other axis is default settings vs. earlier best settings for CountVectorizer. Write a few sentences where you discuss what you see from the table.

Deliveries: Code and results of running the code as described. The 2x2 table. Interpretation of the table.

```
In [20]: arr = np.zeros((2,2))

        create_vs = CountVectorizer, lambda : CountVectorizer(binary=True, ngram_range=(1,2))
```

```
        logreg_params = {"solver": "lbfgs", "max_iter": 500}
        for i, create_clf in enumerate((MultinomialNB, lambda:LogisticRegression(**logreg_para
            for j, create_v in enumerate(create_vs):
                for train_index, test_index in kf.split(kf_features):
                    X_train_text, X_test_text = kf_features[train_index], kf_features[test_in
                    Y_train, Y_test = kf_targets[train_index], kf_targets[test_index]
                    v = create_v()
                    v.fit(X_train_text)
                    X_train, X_test = v.transform(X_train_text), v.transform(X_test_text)

                    clf = create_clf()
                    clf.fit(X_train, Y_train)
                    acc = clf.score(X_test, Y_test)
                    arr[i, j] += acc

        arr /= k
        arr
```

```
Out[20]: array([[0.80611111, 0.85333333],
                [0.84611111, 0.87722222]])
```

```
In [21]: pd.DataFrame(data=arr, index=["Multinomial NB", "Logistic regression"], columns=["defa
```

```
Out[21]:                      default vectorizer  best vectorizer
         Multinomial NB                 0.806111         0.853333
         Logistic regression            0.846111         0.877222
```

1.4 The end

To fullfill a series of experiments, we would normally choose the best classifier after the development stage and test it on the final test set. But we think this suffice for this mandatory assignment. Moreover, we would like to run some more experiments in the future on the development data, before we contaminate them.