Lab2

May 30, 2018

1 SD-TSIA214 Machine Learning For Text Mining - Lab 2

1.1 Sentiment Analysis

1.2 PART ONE - Classifier Implementation

The goal of this TP is to implement a Naive Bayes classifier to perform sentiment analysis on movies, trying to predict if they will have a positive or negative review. Our dataset is composed by 2000 review documents, each one labelled either as positive or negative.

```
In [1]: from glob import glob
        from sentimentanalysis import NB
        import numpy as np
        import os.path as op
        import re

        from nltk.stem.snowball import SnowballStemmer
        from nltk import pos_tag

        from sklearn.base import BaseEstimator, ClassifierMixin
        from sklearn.feature_extraction.text import CountVectorizer
        from sklearn.model_selection import cross_val_score
        from sklearn.naive_bayes import MultinomialNB
        from sklearn.pipeline import Pipeline
        from sklearn.svm import LinearSVC
```

1.2.1 1. Complete the count_words function that will count the number of occurrences of each distinct word in a list of string and return vocabulary (the python dictionary. The dictionary keys are the different words and the values are their number of occurences).

In the *count_words* function implementation, there are fundamentally two choices: get immediately all distinct words present in the collection of texts and then update their count, or start with an empty vocabulary and update it each time a new word is encountered. The second approach was followed. The function, in fact, starts with an empty vocabulary and starts counting words: if the current word is already known, it updates the count of this word in the current text; if the word is not already known, it appends it at the end of current's text count array and adds a new rule to the vocabulary, corresponding to the same index. In the end, zero padding has to be applied to arrays to return a NxN matrix. With respect to the returned **vocabulary** dict, I found a little contrast between this question's specifications on the subject and the comments at the beginning of the method provided. The specifications at the beginning of the method were followed, so vocabulary is a dictionary containing known words as key and as value the index that corresponds to the position of word count in the counts array.

```
In [2]: def count_words(texts, stop_words=None):
            """Vectorize text : return count of each word in the text snippets

            Parameters
            ----------
            texts : list of str
                The texts

            Returns
            -------
            vocabulary : dict
                A dictionary that points to an index in counts for each word.
            counts : ndarray, shape (n_samples, n_features)
                The counts of each word in each text.
                n_samples == number of documents.
                n_features == number of words in vocabulary.
            words = set()
            for text in texts:
                words = words.union(set(text.split()))
            n_features = len(words)
            n_samples = len(texts)
            vocabulary = dict(zip(words), range(n_features))
            counts = np.zeros((n_samples, n_features))
            for k, text in enumerate(texts):
                counts[k][vocabulary[w]] += 1
            """

            # Initialize
            vocabulary = {}
            counts = []
            j=0
            # Remove reduntant stop words
            sw = list(set(stop_words)) if stop_words is not None else []
            # Counts words in each text.
            # New words encountered are appended to the end of the array
            for text in texts:
                # Split text in words
                words = re.split(' |; |, |\n', text)
                # Initialize text vocabulary to known words
                single_count = [0]*len(vocabulary)
                for word in words:
                    # Check for stop words
                    if word not in sw:
```

```
                if word in vocabulary:
                    single_count[vocabulary[word]] += 1
                else:
                    # Add new word both in vocabulary and in current result
                    single_count.append(1)
                    vocabulary[word] = j
                    j = j+1
        counts.append(single_count)

    # Pad arrays with zeros (first arrays don't contain all words)
    countsout = np.zeros((len(texts), len(vocabulary.keys())), dtype=np.int32)
    i=0
    for c in counts:
        countsout[i, :len(c)] = c
        i = i+1
    return vocabulary, countsout
```

1.2.2 2. Explain how positive and negative classes have been assigned to movie reviews (see poldata.README.2.0 file)

According to what is written in the file, reviews have a positive label if they have a rating higher or equal to 3.5 stars out of 5 or 4 out 5, and a grade not lower than B. Reviews have a negative review if their rating is lower than 2/5 stars, or 1.5/4 stars, or a grade equal or lower than C.

1.2.3 3. Complete the NB class to implement the Naive Bayes classifier

Complete the NB class means to carry out the implementation of *fit* and *predict* methods.

The *fit* method computes the probabilities for each class (which is the **prior** probability) simply as Nc = number of texts having class c / N = all texts and the probabilities for each word w conditioned to class c (which the **conditional** probability), representing the probability of having word w in a text knowing that its class is c.

The *predict* method carries out the predicting task by computing a score for each class c: the class with the higher score will then be choosen as the prediction. The score of each class is initialized to the prior probability of that class, and then incremented by the logarithm of the conditional probability for that class of each word present in the text to predict.

```
In [3]: class NB(BaseEstimator, ClassifierMixin):

            def __init__(self):
                self.num_classes = None
                self.classes = {}
                self.prior = {}
                self.condprob = {}

            def fit(self, X, y):
                # Compute number of docs
                N = len(y)
                # Compute classes
```

```
        self.classes = np.unique(y)
        self.num_classes = len(self.classes)

        for c in self.classes:
            # Compute prior probability
            # Take only wordcounts of data with class c
            useful_data = np.array([ X[idx, :] for idx, val in enumerate(y) if val == c
            Nc = len(useful_data)
            self.prior[c] = Nc/N
            # Compute conditional probability
            totcount = {}
            # Sum columns
            totcount[c] = np.array([ sum(x) for x in zip(*useful_data) ])
            # Compute Laplacian
            lapl = np.sum(totcount[c]) + len(totcount[c])
            # Smooth
            self.condprob[c] = (totcount[c]+1)/lapl
        return self

    def predict(self, X):
        sc = np.zeros((X.shape[0], self.num_classes))
        for c in self.classes:
            # Initialize scores sc to prior probability
            sc[:, int(c)] = np.log(self.prior[c])
            # Compute log value
            cp = np.log(self.condprob[c])
            for x in range(X.shape[0]-1):
                # Add log value if word is present in the text
                sc[x, int(c)] = sum([ val for idx, val in enumerate(cp) if X[x][idx] !=
        return [ np.argmax(sc[x]) for x in range(X.shape[0]) ]

    def score(self, X, y):
        return np.mean(self.predict(X) == y)
```

1.2.4 4. Evaluate the performance of your classifier in cross-validation 5-folds.

In [4]:
```
# Load data
filenames_neg = sorted(glob(op.join('.', 'data', 'imdb1', 'neg', '*.txt')))
filenames_pos = sorted(glob(op.join('.', 'data', 'imdb1', 'pos', '*.txt')))

texts_neg = [open(f).read() for f in filenames_neg]
texts_pos = [open(f).read() for f in filenames_pos]
texts = texts_neg + texts_pos
y = np.ones(len(texts), dtype=np.int)
y[:len(texts_neg)] = 0.

print('Loaded data...')
```

```
# Split processed data in train and test data
# Random permutation to split the data randomly
indices = np.random.permutation(len(texts))
size = int(len(texts)/2)
X = texts
X_train = [ X[i] for i in indices[:-size] ]
y_train = y[indices[:-size]]
X_test = [ X[i] for i in indices[-size:] ]
y_test = y[indices[-size:]]

# Load stop words
stop_words = open('./data/english.stop').read().split()
print('Loaded stop words...')
```

```
Loaded data...
Loaded stop words...
```

```
In [5]: vocabulary, X_proc = count_words(X, stop_words)

        nb = NB()
        score = cross_val_score(nb, X_proc, y, cv=5)
        print('With stop words...')
        print('Cross Validation 5-fold score: ' + str(score.mean()) + ' mean ' + str(score.std
```

```
With stop words...
Cross Validation 5-fold score: 0.8215 mean 0.015049916943292404 std
```

1.2.5 5. Change the count_words function to ignore the "stop words" in the file data/english.stop. Are the performances improved ?

```
In [6]: vocabulary, X_proc = count_words(X)

        nb = NB()
        score = cross_val_score(nb, X_proc, y, cv=5)
        print('Without stop words...')
        print('Cross Validation 5-fold score: ' + str(score.mean()) + ' mean ' + str(score.std
```

```
Without stop words...
Cross Validation 5-fold score: 0.8275 mean 0.008366600265340763 std
```

It seems that not using stop words helps - even if by little - in increasing accuracy. In both cases, an accuracy higher than 0.8 is a very satisfying result.