Assignment 2

Skicka in uppgift igen

Inlämningsdatum 29 nov 2020 av 23.59 **Poäng** 10 **Lämnar in** en filuppladdning
Tillgänglig 16 nov 2020 kl 17:00–31 jan kl 23.59 3 månader

Assignment 2: Text Classification with Naive Bayes

In this assignment, you will implement the Naive Bayes classification method and use it for sentiment classification of customer reviews.

Work in groups of one, two or three and solve the tasks described below. Write a short report containing your answers, including the plots and create a zip file containing the report and your Python code.

Alternatively, write a Jupyter notebook including your code, plots, and comments. In this case, when you are finished editing, re-run all the cells to make sure they work and then convert your notebook into a pdf (using the print function). Submit both the .ipynb file and the .pdf file.

Submit your solution through the Canvas website.

In this assignment, the Naive Bayes code must be your own. **You are not allowed to use machine learning toolkits such as scikit-learn for the Naive Bayes part, except in one of the optional tasks.**

Deadline: November 29

Didactic purpose of this assignment:

- understand Naive Bayes, one of the simplest probabilistic classification techniques;
- see how a non-trivial statistical model is implemented in practice;
- practice your Python coding skills for common tasks such as computing frequencies

Preliminaries

Read up the Naive Bayes classifier: how to compute apply the Naive Bayes formula, and how to estimate the probabilities you need. For instance, repeat **the slides** from the lecture about estimation and Naive Bayes. In particular the slides which describe the Naive Bayes formula, and the ones on how to estimate the required probabilities.

If you wish, you may also have a look at the following classic paper:

- **Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan***: Thumbs up? Sentiment Classification using Machine Learning Techniques (https://dl.acm.org/doi/pdf/10.3115/1118693.1118704)*. In Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing

Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002).

The data set we are using was originally created for the experiments described in the following paper. The research described here addresses the problem of *domain adaptation*, such as adapting a classifier of book reviews to work with camera reviews.

- **John Blitzer, Mark Dredze, and Fernando Pereira**: ***Biographies, Bollywood, Boom-boxes and Blenders: Domain Adaptation for Sentiment Classification.*** ***(http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=F5E379EDCCDBD1E4055423EAD594A6E0?doi=10.1.1.143.7660&rep=rep1&type=pdf)*** In Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL 2007).

Preparatory remarks

Frequency-counting in Python. The built-in data structure called `Counter` is a special type of Python dictionary that is adapted for computing frequencies. In the following example, we compute the frequencies of words in a collection of two short documents. We use `Counter` in three different ways, but the end results are the same (`freqs1`, `freqs2`, and `freqs3` are identical at the end). The `Counter` will not give a `KeyError` if you look for a word that you didn't see before.

```
from collections import Counter

example_documents = ['the first document'.split(), 'the second document'.split()]

freqs1 = Counter()
for doc in example_documents:
    for w in doc:
        freqs1[w] += 1

freqs2 = Counter()
for doc in example_documents:
    freqs2.update(doc)

freqs3 = Counter(w for doc in example_documents for w in doc)

print(freqs1)
print(freqs1['the'])
print(freqs1['neverseen'])
```

Logarithmic probabilities. If you multiply many small probabilities you may run into problems with numeric precision: the probability becomes zero. To handle this problem, I recommend that you compute the *logarithms* of the probabilities instead of the probabilities. To compute the logarithm in Python, use the function `log` in the `numpy` library.

The logarithms have the mathematical property that `np.log(P1 * P2) = np.log(P1) + np.log(P2)`. So if you use log probabilities, all multiplications (for instance, in the Naive Bayes probability formula) will be replaced by sums.

If you'd like to come back from log probabilities to normal probabilities, you can apply the exponential function, which is the inverse of the logarithm: `prob = np.exp(logprob)`. However, if the log probability is too small, `exp` will just return zero.

Reading the review data

Download **this file**. This is a collection of customer reviews from six of the review topics used in the paper by Blitzer et al., (2007) mentioned above. The data has been formatted so that there is one review per line, and the texts have been split into separate words ("tokens") and lowercased. Here is an example of a line.

```
music neg 544.txt i was misled and thought i was buying the entire cd and it contains one song
```

A line in the file is organized in columns:

- 0: topic category label (books, camera, dvd, health, music, or software)
- 1: sentiment polarity label (`pos` or `neg`)
- 2: document identifier
- 3 and on: the words in the document

Here is some Python code to read the entire collection[1].

```
from codecs import open
from __future__ import division

def read_documents(doc_file):
    docs = []
    labels = []
    with open(doc_file, encoding='utf-8') as f:
        for line in f:
            words = line.strip().split()
            docs.append(words[3:])
            labels.append(words[1])
    return docs, labels
```

We first remove the document identifier, and also the topic label, which you don't need unless you solve the first optional task. Then, we split the data into a training and a validation part. For instance, we may use 80% for training and the remainder for validation as follows.

```
all_docs, all_labels = read_documents('all_sentiment_shuffled.txt')

split_point = int(0.80*len(all_docs))
train_docs = all_docs[:split_point]
train_labels = all_labels[:split_point]
val_docs = all_docs[split_point:]
val_labels = all_labels[split_point:]
```

Estimating parameters for the Naive Bayes classifier

Write a Python function that uses a training set of documents to estimate the probabilities in the Naive Bayes model. Return some data structure containing the probabilities or log probabilities. The input parameter of this function should be a list of documents and another list with the corresponding polarity labels. It could look something like this:

```
def train_nb(documents, labels):
    ...
    (return the data you need to classify new instances)
```

Hint 1. In this assignment, it is acceptable if you assume that we will always use the `pos` and `neg` categories. However, it is of course nicer if the possible categories are not hard-coded, especially if you solve the last optional task.

Hint 2. Some sort of smoothing will probably improve your results. You can implement the smoothing either in `train_nb` or in `score_doc_label` that we discuss below.

Classifying new documents

Write a function that applies the Naive Bayes formula to compute the logarithm of the probability of observing the words in a document and a sentiment polarity label. `<SOMETHING>` refers to what you returned in `train_nb`.

```
def score_doc_label(document, label, <SOMETHING>):
    ...
    (return the log probability)
```

Sanity check 1. Try to apply `score_doc_label` to a few very short documents; to convert the log probability back into a probability, apply `np.exp` or `math.exp`. For instance, let's consider small documents of length 1. The probability of a positive document containing just the word "great" should be a small number, depending on your choice of smoothing parameter, it will probably be around 0.001–0.002. In any case, it should be higher than the probability of a negative document with the same word. Conversely, if you try the word "bad" instead, the negative score should be higher than the positive.

Sanity check 2. Your function `score_doc_label` should not crash for the document `['a', 'top-quality', 'performance']`.

Next, based on the function you just wrote, write another function that classifies a new document.

```
def classify_nb(document, <SOMETHING>):
    ...
    (return the guess of the classifier)
```

Again, apply this function to a few very small documents and make sure that you get the output you'd

expect.

Evaluating the classifier

Write a function that classifies each document in the validation set and returns the list of predicted sentiment labels.

```
def classify_documents(docs, <SOMETHING>):
    ...
    (return the classifier's predictions for all documents in the collection)
```

Next, we compute the *accuracy*, i.e. the number of correctly classified documents divided by the total number of documents.

```
def accuracy(true_labels, guessed_labels):
    ...
    (return the accuracy)
```

What accuracy do you get when evaluating the classifier on the validation set?

What about the F1 score? Compute and report the F1 score. Do not use the built-in function from scikit-learn.

Comment on the difference between these two evaluation metrics.

Error analysis

Find a few mis-classified examples by selecting a few short documents where the **probabilities** were particularly high in the wrong direction. Show how you made the selection and comment on why you think they were hard to classify.

Cross-validation

We have tried the training-validation split. Now let's try to evaluate the classifier using cross-validation. Split the data set using the following techniques:

- 10-fold cross validation
- Leave-one-out cross validation

Here is a code stub that shows the idea:

```
for fold_nbr in range(N):
    split_point_1 = int(float(fold_nbr)/N*len(all_docs))
    split_point_2 = int(float(fold_nbr+1)/N*len(all_docs))

    train_docs_fold = all_docs[:split_point_1] + all_docs[split_point_2:]
    train_labels_fold = all_labels[:split_point_1] + all_labels[split_point_2:]
    val_docs_fold = all_docs[split_point_1:split_point_2]
```

```
    ...
    (train a classifier on train_docs_fold and train_labels_fold)
    (apply the classifier to val_docs_fold)
...
```

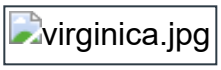
Report the results for both cross validation techniques (i.e. 10-fold and leave-one-out) and comment on how you interpret the results. What's the difference between them?

Note: for leave-one-out validation, you can stop after 100 iterations and report the results.

Naive Bayes for numerical data

Here is a copy of the **famous flower data set** **(https://en.wikipedia.org/wiki/Iris_flower_data_set)** created by **Fisher** **(https://en.wikipedia.org/wiki/Ronald_Fisher)**. It consists of five columns: four numerical columns containing measurements of various parts of the flowers, and one categorical column describing the type of iris. This category is what we will try to predict. (Images below taken from Wikipedia.)

Setosa ***(https://en.wikipedia.org/wiki/Iris_setosa)***



Virginica ***(https://en.wikipedia.org/wiki/Iris_virginica)***

Versico ***(https://en.wikipedia.org***

Now, let's modify your Naive Bayes implementation so that it can handle numerical data, not just words. We will use a model similar to **GaussianNB** **(http://scikit-learn.org/stable/modules/naive_bayes.html#gaussian-naive-bayes)** in scikit-learn.

First, replace the word occurrence probabilities with Gaussian distributions corresponding to the four numerical columns. In `train_nb`, estimate the parameters (mean and standard deviation) of the 12 distributions using maximum likelihood. (That is, just compute sample means and sample variances for each column for each subgroup.) Then also modify `score_doc_label` accordingly. **Hint**: use the log of the pdfs instead of the log of the word occurrence probabilities.

Implement the Gaussian naive Bayes classifier and evaluate it using the accuracy and F1 score. The validation data should be created by 10 fold cross validation. Discuss the result.

The following tasks are all optional tasks for your enjoyment: they are not required for a pass or a VG

(Optional) Computing an interval estimate for the accuracy

This part is probably easiest to do after we've covered interval estimates.

Compute a 95% interval estimate for the accuracy using a method of your choice. You can use the any method we saw during the lecture, bootstrapping, or any of the methods described on **this Wikipedia page (https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval)**.

(Optional) Comparing the accuracy to a given target value

Is your classifier's accuracy significantly different from 0.80 with a *p*-value of at most 0.05? Use the exact binomial test (`scipy.stats.binom_test`) to find out.

(Optional) Comparing two classifiers

For this task, you need two *different* classifiers. For instance, you could train Naive Bayes using two different values for the smoothing parameter. Or you could use a classifier from scikit-learn, see the optional task below.

Carry out a McNemar test and compare the two classifiers (on the 20% validation set or with cross-validation). What is the *p*-value you get for the comparison?

(Optional) Implement a six-category classifier

Implement a classifier that guesses the topic category label instead of the sentiment. (You'll obviously also need to change `read_documents` a little bit.)

(Optional) Learning curve

Set aside 20% of your data as the validation set. Use the remaining data to compute a **learning curve**: select training sets of increasing sizes, e.g 10%, 20%, etc. For each training set size, compute the accuracy on the validation set. Plot the learning curve.

(Optional) Print the informative features

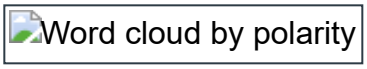
Try to invent a method that finds the features (words) that are most indicative of the positive and the negative categories. To exemplify, *lovely* and *recommend* might be strong positive features and *horrible* and *waste* strong negative. Write a function that prints the *k* strongest features for the respective categories.

```
def print_top_nb(k, <SOMETHING>):
    ...
```

Hint. When I solved this task, I had to set the smoothing parameter to quite a high value to get interpretable results.

If you have nothing better to do, try to make a nice visualization of the words. Maybe some sort of scatterplot where the sizes correspond to frequencies and the colors to the polarity? Another alternative could be to make a word cloud. For instance, Andreas Müller has created **a Python library**

(https://github.com/amueller/word_cloud/) to create word clouds.



Word cloud by polarity *Word cloud by topic*

(Optional) Train a classifier using scikit-learn

Here is an example of how you can train a classifier using scikit-learn:

```
from sklearn.svm import LinearSVC
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import make_pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_selection import SelectKBest

def train_sklearn(docs, labels):
    vec = TfidfVectorizer(preprocessor = lambda x: x,
                          tokenizer = lambda x: x)
    sel = SelectKBest(k='all')
    clf = LinearSVC()
    #clf = MLPClassifier()
    pipeline = make_pipeline(vec, sel, clf)
    pipeline.fit(docs, labels)
    return pipeline
```

This code will train a classifier called *linear support vector classifier*. If you comment out the `LinearSVC` and uncomment the `MLPClassifier` line, you will get a neural network instead. (To get the neural network to work, you probably need to tell the feature selector `SelectKBest` to use a limited number of features: change `'all'` to some number, such as 1000.)

To classify the documents, you can use the following function. It returns an array containing the guessed labels for each document, just like your `classify_documents` above. The input `model` is the object returned by `train_sklearn`.

```
def classify_documents_sklearn(docs, model):
    return model.predict(docs)
```

Here **(http://scikit-learn.org/stable/supervised_learning.html)** is a list of classification methods available in scikit-learn.

(Optional) Improve the features

Try to think of ways to improve the accuracy by changing the features used for classification.

One possible improvement could be to try to do something about negation, since it is obvious that negation is not well handled by the bag-of-words approach we are using. Pang et al., (2002) used this trick: they added a special negation marker to all tokens appearing in a sentence after a negation (*not*, *n't* etc).

Another popular trick for word-based classifiers is to use *bigrams*, or word pairs, in addition to single-word features. This also addresses the problem of negations to some extent. Can you improve your classifier by using bigrams?

(Optional) Domain sensitivity

Select two topic categories from the set of reviews, e.g. camera and book reviews. Create training and validation sets for each of the topics. How much does the accuracy drop when when you apply a camera review classifier to a validation set of book reviews?

[**1**] The purpose of the first two lines is to make the code backwards-compatible with Python 2.
