**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. Write a report containing your answers, including the visualizations. Submit your report and your Python code/notebook.

### **Preliminaries**

Read up the Naive Bayes classifier: how to compute apply the Naive Bayes rule, and how to estimate the probabilities you need.

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*](http://www.aclweb.org/anthology/W/W02/W02-1011.pdf)*.*In Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002).

The dataset 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://aclweb.org/anthology/P/P07/P07-1056.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**](http://www.cse.chalmers.se/~richajo/dit862/data/all_sentiment_shuffled.txt) (URL: <http://www.cse.chalmers.se/~richajo/dit862/data/all_sentiment_shuffled.txt>). 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.

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 an evaluation part. For instance, we may use 80% for training and the remainder for evaluation.

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]

eval\_docs = all\_docs[split\_point:]

eval\_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.

**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 test 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 test set?

### **Error analysis**

Find a few misclassified documents and comment on why you think they were hard to classify. For instance, you may select a few short documents where the probabilities were particularly high in the wrong direction.

### **Cross-validation**

Since our estimation of the accuracy is based on a fairly small set, your interval was quite wide. We will now use a trick to get a more reliable estimate and tighter interval.

In a **cross-validation**, we divide the data into *N* parts (**folds**) of equal size. We then carry out *N* evaluations: each fold once becomes a test set, while the other folds form the training set. We then combine the results of the *N* different evaluations. This trick allows us to get results for the whole dataset, not just a small test set.

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:]

eval\_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 eval\_docs\_fold)

...

(combine the outputs of the classifiers in all folds)

Implement the cross-validation method. Then estimate the accuracy and compute a new interval estimate. A typical value of *N* would be between 4 and 10.

Reference: This is homework was copied from the following website: <http://www.cse.chalmers.se/~richajo/dit862/assignment2.html>. You may check the original materials for details.