Programming Assignment 4 – Basic Probability, Computing and Statistics 2017

Jakub Dotlačil and Bas Cornellisen

Submission deadline: Wednesday, May 2nd, 8 p.m.

Note: if the assignment is unclear to you or if you get stuck, do not hesitate to contact Jakub or Bas. You can also post a question on Canvas.

1 Assignment

This week we will implement a Naïve Bayes (NB) classifier for text data. If you do not remember what a NB classifier is, check here. We will first train the classifier and then make predictions based on our estimates.

1.1 Naïve Bayes Terminology

In NB training we assume i.i.d. data points (x_i, y_i) $(1 \leq i \leq n)$ where x_i is a k-dimensional vector and y is 1-dimensional. The entries of x are called **features**. When we have to make predictions we are presented with vectors x_t $(t > n)$ which have no accompanying y-values. Our task is to infer these y-values. The posterior probability of a particular value for Y_t according the the Naïve Bayes assumption is

$$P(Y_y = y_t | X_t = x_t) \propto P(Y_t = y_t) \prod_{j=1}^{k} P(X_{tj} = x_{tj} | Y_t = y_t)\,. \tag{1}$$

By convention, we call the Y-variables **labels**, **classes** or **categories**. We will use these terms interchangeably to get you used to them. A Naive classifier thus performs inference based on k features (where k may differ across data points). For the purpose of this exercise, we take the features to be words and the labels to be names of news groups.

1.2 Data

We will use the famous 20 newsgroups data set for this exercise. As the name suggest, there are 20 groups of text and our task is to find the right group for given words. This implies a baseline performance achieved by randomly choosing a group of 5%. The data are emails within newsgroups collected in the 1990's.

For a start, we will simply use all strings separated by whitespace as features for our NB model. We call these strings words.

1.3 Smoothing

Many words will only occur in one class c. When we see a text from that class during prediction time, the other classes will not have probabilities for as many words as c. There are two ways to go about this:

1. Simply ignore the words that have 0 probability under a class. This will lead to many classes having fewer terms in the NB product than c. Thus, these classes will have a higher posterior.
2. Do model all words for all classes. Unfortunately, most unseen texts (the ones we are dealing with at prediction time) contain for each class at least one word that has 0 probability. In effect, the posterior probability for all classes will be 0. Notice that this means the posterior is not even defined as it does not sum up to 1.

As you can see, both ways will give us the wrong results. There is a cleverer way, however. We remember all words that we have encountered during training. These define our vocabulary. During prediction we will ignore all words not in the vocabulary. Moreover, we perform smoothing. This means that, before running any predictions, we add a constant count for *all* words in the vocabulary to the training counts of every class. This has the effect that all vocabulary words will have positive probability under all classes.

1.4 Logprobs

There is one more practical problem we have to solve. When you classify a document, you have to compute a potentially large product of probabilities. These products can become very, very small. In fact, they become so small that our computers won't be able to represent them in memory and just turn them into 0.

The standard way to avoid this problem is to work with logarithms of probabilities (logprobs) instead. You should always use logprobs when performing probabilistic computations! Multiplication and division of probabilities is then straightforward because for positive numbers $a, b > 0$ it holds that

$$\begin{aligned}\log(a \cdot b) &= log(a) + log(b)\\ \log\left(\frac{a}{b}\right) &= log(a) - log(b)\\ \log\left(a^b\right) &= \log(a) \cdot b\end{aligned}$$

Addition and subtraction is slightly more involved. Let $c = \log(a)$ and $d = \log(b)$. The naïve way to do addition would be through simple exponentiation.

$$\log(\exp(c) + \exp(d)) = \log(a + b)$$

This is inefficient, however, since we need to compute two exponentials and one logarithm. Computers are slow at that. Instead, the following equivalence is exploited. Without loss of generality, we also assume that $c > d$.

$$\begin{aligned}\log(\exp(c) + \exp(d)) &= \log(\exp(c) \cdot (1 + \exp(d - c))) \\ &= c + \log(1 + \exp(d - c))\end{aligned}$$

There are several advantages to using this trick. First, we only compute one exponential and one logarithm. Second, the logarithm computation is already implemented in many programming languages (including Python) as a function called `log1p` (see here for documentation). The `log1p` function computes $\log(1 + \exp(d - c))$ very efficiently when the exponent is small.

To sum up, you have to avoid turning that joint probabilities turn to `0`, and to do so, you have to work with log-probs. Make sure to transform probabilities to log-probs before doing any predictions.

1.5 Your Task

We have implemented a commandline interface and a skeleton of the Naive Bayes class for you. Both can be found here. Do not manipulate `naive_bayes_classifier.py`. Your task is to only work on the methods of the class NaiveBayes, which is in the file naive_bayes.py. Please implement the methods that are marked with a TODO. For prediction, please output the a posteriori most likely label. Feel free to add any methods and data structures that you deem necessary.

Challenge yourself On the development set, which we provide together with the code, you should achieve an accuracy of 81.05% if the smoothing constant is set to 1. Can you do better than that? Try different smoothing constants or try to change the words (e.g. by lowercasing them) or try to include more features (characters, for example). The possibilities are limitless!

1.6 Running the Code

You will have to supply commandline arguments to the script this time. This can be done in Pycharm. Right-click on the run symbol and select *Add parameters*. In the field *script parameters* you then have to enter the following for `naive_bayes_classifier.py`

```
--training-corpus-dir <Path to 20news-18828>
   --test-set-directory <Path to dev-set>
```

(Note: the line break was inserted by LaTeXand is not needed in Pycharm.) You can optionally add

```
--keys <Path to dev_key.txt>
```

in which case the script will also run an evaluation of your output. For this to work you have to assign a string containing your Python command (either `python` or `python3` for most of you) to the variable `my_python` at the top of the file `naive_bayes_classifier.py`.

The output will be a file called `predictions.txt`. You can also separately run an evaluation of your output using the accuracy checker (which is the same script used if you provide the `--keys` argument above). For that, you will have to pass argument to `accuracy_checker.py`, as well. These are simply

```
<Path to predictions.txt> <Path to dev_keys.txt>
```

2 Grading

You will be graded on a test set that we will make available during peer review. The test set is structured like the dev set but contains different files.

2 points	The code produces predictions that are in the correct format and assign labels that it has been trained on (instead of arbitrarily named labels)
2 point	The test-set (not dev-set(!)) accuracy is at least 20%.
2 point	The test-set (not dev-set(!)) accuracy is at least 40%.
2 point	The test-set (not dev-set(!)) accuracy is at least 60%.
1 point	The test-set (not dev-set(!)) accuracy is at least 70%.
1 point	The test-set (not dev-set(!)) accuracy is higher than 80%.