

# Programming 3

[Submit Assignment](#)

---

<b>Due</b> Thursday by 11:59pm	<b>Points</b> 100	<b>Submitting</b> a file upload
<b>File Types</b> zip and py	<b>Available</b> until Nov 20 at 11:59pm	

---

## COMS W4701 - Artificial Intelligence - Programming Homework 3

### Naive Bayes Spam Classifier

**Due: 11:59pm on Thursday, November 16th**

**Total Points: 100**

You are welcome to discuss the problems with other students but you must turn in your own work. Please review the academic honesty policy for this course (at the end of the syllabus page).

Create a .zip or .tgz archive containing any of the files you submit. Upload that single file to Courseworks. The file you submit should use the followign naming convention:

YOURUNI\_programming3.[zip | tgz]. For example, my uni is db2711, so my file should be named db2711\_programming3.zip or db2711\_programming3.tgz.




As a reminder, any assignments submitted late will incur a 20 point penalty. No submissions will be accepted later than 4 days after the submission deadline.

The instructions below are fairly specific and it is okay to deviate from implementation details. **However: You will be graded based on the functionality of each function/method. Make sure the function signatures (names, parameter and return types/data structures) match exactly the description in this assignment.**

## Introduction

In this assignment you will implement a Naive Bayes classifier and use it for spam filtering on a text message data set.

### Data Set

The data set you will train and evaluate your classifier on contains 5574 SMS text messages, 747 of which are spam. You will use three files, [train.txt](#) , [dev.txt](#) , and [test.txt](#)  containing training, development, and testing data respectively.

The raw data was downloaded from <http://dcomp.sor.ufscar.br/talmeida/smsspamcollection/> (<http://dcomp.sor.ufscar.br/talmeida/smsspamcollection/>). There is also a paper describing the data set:



Almeida, T.A., Gómez Hidalgo, J.M., Yamakami, A. **"Contributions to the Study of SMS Spam Filtering: New Collection and Results"**. Proceedings of the 2011 ACM Symposium on Document Engineering (DOCENG'11).

### Data File Format

Each line in a data file represents one input/output sample. A line contains two fields, separated by a tabstop ("t"). The first field contains the output label, "spam" or "ham". The second field contains the actual message text. For example

```
spam    Congratulations ur awarded either £500 of CD gift vouchers & Free entry 2 our £100 weekly draw txt MUSIC to
87066 TnCs www.Ldew.com 1 win150ppmx3age16
ham     Just hoping that wasn't too pissed up to remember and has gone off to his sisters or something!
```

## Getting Started

Download the file [homework3.py](#)  . You will complete the class `NbClassifier` in this file, which represents a Naive Bayes classifier.

You will eventually run this file as follows, to train a model on the training data and test it on the validation data.

```
$ python homework3.py train.txt dev.txt
Precision:0.0 Recall:0.0 F-Score:0.0 Accuracy:0.0
```

Obviously, instead of 0.0, your program should eventually print the actual results once you are done with this problem.

## Part 1 - Extracting Attributes

The attributes you will use for the classifier will be individual words in the text.

- First, write the function `extract_words`, which should read in the text of an SMS message and return a list of individual words (strings). Pre-processing is fairly important for text classification tasks. For now, simply lower-case the text, remove any punctuation, and then split it on white spaces. Make sure to remove any additional leading and trailing white spaces per word. We will improve the pre-processing step later.
- Next, write the method `collect_attribute_types`, which should compute a vocabulary consisting of the set of unique words occurring at least  $k$  times in the training data. Start by setting  $k$  to 1. You can tune the value of  $k$  later on the validation set. The method should return nothing, but it should populate the `self.attribute_types` attribute of the `NbClassifier` object.

## Part 2 - Training the Classifier

Training a Naive Bayes classifier amounts to estimating two probability distributions from the training data.

1. The prior for the labels, i.e.  $P(\text{Label}=y)$  for each label  $y$ .
  2. The conditional probability for each attribute, given the label, in our case  $P(\text{Word}=w \mid \text{Label}=y)$  for each label  $y$  and word  $w$  that occurs in the vocabulary, i.e. the `self.attribute_types` set.
- Write the method `train`, which should read in the training data (using `extract_words` to preprocess it) and then populate the `self.label_prior` and `self.word_given_label` dictionaries. The keys for `label_prior` should just be the two labels (i.e. "ham" or "spam"). The keys for `word_given_label` should be (word, label) tuples, for example ('congratulations','spam').
  - Compute the conditional probability as
$$P(\text{Word}=w \mid \text{Label}=y) = \text{count}(w,y) / [\text{total number of words in any training example with label } y].$$
The intuition here is that each word is generated by its own event.

- Unfortunately, not all word/label combinations appear in the training data. Your conditional probability estimates should be "smoothed". You can use the following simple additive smoothing approach (also known as Laplacian Smoothing).

$$P(\text{Word}=w|\text{Label}=y) = (\text{count}(w,y)+c) / ([\text{total number of words in any training example with label } y] + c * \text{vocabulary size}).$$

The idea is to add  $c$  more occurrence for each word/label combination, giving a little probability mass to unseen combinations. Start by setting  $c$  to one.

## Part 3 - Testing the Classifier

- Write the method `predict` which should take the text of an SMS as input (as a string). We want to compute the probability for each label given the words in the string as:

$$P(y | \text{word1}, \text{word2}, \dots, \text{wordn}) = \alpha * P(y) * P(\text{word1} | y) * P(\text{word2} | y) * \dots * P(\text{wordn} | y)$$

where  $\alpha$  is a normalization factor to make sure that the probabilities for all labels sum up to 1 (see slides).

Because we are only interested in classification, your function can just compute the joint probability of the label and words:

$$P(y, \text{word1}, \text{word2}, \dots, \text{wordn}) = P(y) * P(\text{word1} | y) * P(\text{word2} | y) * \dots * P(\text{wordn} | y)$$

Unfortunately, for long inputs, the probabilities become extremely small. To be able to still compare the two labels we can run the computation in log space. Instead of the expression above, compute the following:

$$\log(P(y, \text{word1}, \text{word2}, \dots, \text{wordn})) = \log(P(y)) + \log(P(\text{word1} | y)) + \log(P(\text{word2} | y)) + \dots + \log(P(\text{wordn} | y)).$$

You might want to import the `math` module and use the `math.log` functions.

The function should then return a dictionary mapping the label to the log probability for that label. Here is an example in which the input would be classified as spam:

```
{"ham":-197.80923461412195, "spam":-158.7005945688615}
```



- Write the method `evaluate`, that takes the filename for the validation or test set as parameters and returns (a tuple of) four values: precision for predicting spam, recall for predicting spam, fscore, classification accuracy.  
In the function, call the `predict` method for each input sample, and then count how many true positive false positive, true negatives, and false negative predictions are made (with respect to predicting "spam").
- Run your classifier on the validation set `dev.txt`. You should achieve accuracies in the high 90% range and an F-score in the low 0.90s.

## Part 4 - Tuning the Classifier

The classifier has at least two different parameters that can be tuned: the smoothing parameter  $c$  and the word cutoff  $k$  (see part 1). Experiment (by hand) with these parameters on the validation set until you achieve better performance. Your final submission should include a comment at the beginning of the file, stating your original performance (with the defaults) on the validation set and your improved performance after tuning on

both validation and test.

## Part 5 - Improving Pre-Processing

Our classifier contains features for words like "to", "it", or "and", (so-called *stop words*), which occur frequently in any English text and are probably not very informative for distinguishing between the two classes. A common improvement for Naive Bayes text classification is to remove such stop-words from the feature set. The file [stopwords\\_mini.txt](#)   contains a very short list of stopwords. Modify your code so the stopwords list is read in when NbClassifier is instantiated and stopwords are excluded from the feature set. Do not modify the function or method signatures.

You can modify the list of stopwords to improve performance on the dev set.

Many other improvements are possible (this part is optional). For example, you could replace all numbers with a common symbol. It might also be useful to leave certain types of punctuation in the feature set (for example \$, or emoticons :-)).