# Task 2: Naive Bayes Classifier

In the previous task we only used selected words for the classification, and we relied on the manual work that went into the lexicon to decide which words were likely to be in reviews with a particular sentiment. The sentiment lexicon is general: it is supposed to be applicable to any English text. We will now create a statistical classifier, learning which words are positive and negative in movie reviews. Unlike the lexicon approach, this classifier uses **all** of the words in the text.

We recommend that you read the first 7 pages of Jurafsky and Martin's draft chapter on Naive Bayes for more information.

## Step 0: Data preparation.

Use the dataset from Task 1 for today's training and development. We split the 1800 document dataset from Task 1 so there are 200 documents in your 'development' (also called 'validation') set, and 1600 in the training set. Both of these sets are balanced (i.e. they contain the same number of positive and negative reviews.) The code to make the split has been provided in `utils/sentiment_detection/data_loader.py`, specifcally in the method `split_data(review_data, seed)`. There are a further 200 reviews that you do not have access to at the moment (held-out data) that your model will be evaluated on when you submit your code to the automatic grader.

Given these splits, you can rerun the simple classifer from Task 1 on the 200 examples in the development set. Later you can see whether you get an improvement with the Naive Bayes classifier you implement today. Everything you need to implement this step can be found within `tick2.py`. Familiarise yourself with the execution pipeline first.

## Step 1: Parameter estimation.

Your first task in developing the Naive Bayes classifier is to estimate probabilities from observations of the data, namely: the probabilities of the classes in the dataset $P(\text{POS})$, $P(\text{NEG})$; and the probabilities of each word in the dataset appearing with a particular class label $P(\text{word}|\text{POS})$ and $P(\text{word}|\text{NEG})$.

Write a program that estimates these probabilities from the training data. Then compute the logs of these probabilities.

## Step 2: Classification.

We will now use the probabilities to apply the classification argmax formula of Naive Bayes to the development set. We predict which class (POS or NEG) to assign a review to, as

$$c_{NB} = \underset{c \in \{\text{POS,NEG}\}}{\arg\max} \Big\{ \log P(c) + \sum_{i \in positions} \log P(w_i|c) \Big\}$$

where *positions* is the set of indexes into the all words in the document. $P(c)$ is the probability of a document belonging to a given class. For a balanced data set, like the one here, this probability is the same for each class (i.e. 0.5) but this will not always be the case. The argmax formula gives you a decision (a classification) for each document, either negative or positive.

How well did this program perform? As before, we can compare the system's performance to the truth but we should use only the development set to do this.

## Step 3: Smoothing.

Have you noticed any issues when using the log probabilities calculated in Step 1?

When using a Naive Bayes classifier, you need to consider what to do with unseen words – words which occur in the development/test data but which do not occur in the training data at all. You will also need to consider what to do with words which were seen only in one class of document, either positive or negative ones. Write down possible solutions to this problem.

Modify your calculation of log probabilities to implement add-one (Laplace) smoothing – add one to all the counts.

> Example: Say a word *jabberwocky* appeared in your training set only once, in a positive review. You should count the following counts:

| smoothing | positive | negative |
|---|---|---|
| none | 1 | 0 |
| add-one | 2 | 1 |

*Software engineering advice:* instead of copying your code from the unsmoothed version then modifying it, it is better practice to create an auxillary function (e.g. `def calculate_log_probabilities( ... ): ...`) which takes a boolean `smoothed` argument, then calling this function in both `calculate_smoothed_log_probabilities` and `calculate_unsmoothed_log_probabilities` (setting `smoothed` to true or false as appropriote). Doing this avoids code duplication, which means you

won't have to make changes in multiple places if you find an issue.

How does your classifier perform now?

Once you have successfully developed and tested a system which incorporates add-one smoothing, you may submit your code to the online tester. Since your simple classifier from Task 1 is used as well, submit both `tick1.py` and `tick2.py`. Once your code has passed, you may contact a demonstrator to obtain Tick 2.