Problem 3. [67] A Naive Bayes Classifier for News Article Classification

One application of Naive Bayes classifiers is for classifying news articles into various categories. In this homework you are to implement a general Naïve Bayes classifier for categorizing news articles as either SPORTS or BUSINESS (i.e., not SPORTS). The dataset we are providing to you consists of news articles (http://mlg.ucd.ie/datasets/bbc.html) from a BBC news dataset that have been labeled as either SPORTS or BUSINESS, and which we have split into a training set and a testing set. If you open the files, you will see that the first word in each line is the class label and the remainder of the line is the news article.

Methods to Implement

We have provided code for you that will open a file, parse it, pass it to your classifier, and output the results. What you have to focus on is the implementation in the file `NaiveBayesClassifierImpl.java` If you open that file, you will see the following methods that you must implement:

- `void train(Instance[] trainingData, int v)`
  This method should train your classifier using the given training data. The integer argument `v` is the size of the total vocabulary in your model. Store this argument as a field because you will need it in computing the smoothed class-conditional probabilities. (See the section on Smoothing below.)
- `void documents_per_label_count(Instance[] trainingData)`
  This method should count the number of documents per class label in the training set. The format of the printout is one line for each class label.
- `void words_per_label_count(Instance[] trainingData)`
  This method should count the number of words per label in the training set. The format of the printout is one line for each label.
- `double p_l(Label label)`
  This method should return the prior probability of the label in the training set. In other words, return *P*(`SPORTS`) `if label == Label.SPORTS` or *P*(`BUSINESS`) `if label == Label.BUSINESS`
- `double p_w_given_l(String word, Label label)`
  This method should return the conditional probability of `word` given `label`. That is, return *P*(*word*|*label*). To compute this probability, you will use smoothing. Read the note below on how to implement this.
- `ClassifyResult classify(String[] words)`
  This method returns the classification result for a single news article.
- `public ConfusionMatrix calculateConfusionMatrix(Instance[] testData)`
  This method takes a set of test instances and calculates a confusion matrix. The return type is `ConfusionMatrix`. The member variables of that class should be self explanatory. The following table explains the different cells of the confusion matrix:

	True Sports	**True Business**
Classified Sports	True Positive (TP)	False Positive (FP)
Classified Business	False Negative (FN)	True Negative (TN)

We have also defined four class types to assist you in your implementation. The `Instance` class is a data structure holding the label and the news article as an array of words:

```
public class Instance {
        public Label label;
        public String[] words;
}
```

The `Label` class is an enumeration of our class labels:

```
public enum Label { SPORTS, BUSINESS }
```

The `ClassifyResult` class is a data structure holding a label and two log probabilities (whose values are described in the log probabilities section below):

```
public class ClassifyResult {
         public Label label;
         public double log_prob_sports;
         public double log_prob_business;
}
```

The `ConfusionMatrix` class is a data structure to be instantiated in the `calculateConfusionMatrix` method with True Positive(TP), False Positive(FP), False Negative(FN) and True Negative(TN) numbers for the classification task on the test data:

```
public class ConfusionMatrix { int TP, FP, FN, TN; }
```

The *only* provided file you are allowed to edit is `NaiveBayesClassifierImpl.java` but you are allowed to add extra helper class files if you like. Do not include any package paths or external libraries in your program. Your program is only required to handle binary classification problems.

Smoothing

There are two concepts we use here

- *Word token*: an occurrence of a given word
- *Word type*: a unique word as a dictionary entry

For example, “the dog chases the cat” has 5 word tokens but 4 word types; there are two tokens of the word type “the.” Thus, when we say a word “token” in the discussion below, we mean the number of words that occur and *NOT* the number of unique words. As another example, if a news article is 15 words long, we would say that there are 15 word tokens. For example, if the word “lol” appeared 5 times, we say there were 5 tokens of the word type “lol.”

The conditional probability $P(w|l)$, where w represents some word token and l is a label, is a multinomial random variable. If there are $|V|$ possible word types that might occur, imagine a $|V|$-sided die. $P(w|l)$ is the likelihood that this die lands with the w-side up. You will need to estimate two such distributions: $P(w|Sports)$ and $P(w|Business)$.

One might consider estimating the value of $P(w|Sports)$ by simply counting the number of w tokens and dividing by the total number of word tokens in all news articles in the training set labeled as *Sports*, but this method is not good in general because of the “unseen event problem,” i.e., the possible presence of events in the test data that did *not* occur at all in the training data. For example, in our classification task consider the word “foo.” Say “foo” does *not* appear in our training data but *does* occur in our test data. What probability would our classifier assign to $P(foo|Sports)$ and $P(foo|Business)$? The probability would be 0, and because we are taking the sum of the logs of the conditional probabilities for each word and log 0 is undefined, the expression whose maximum we are computing would be undefined. This wouldn’t be a good classifier.

What we do to get around this is we pretend we actually *did* see some (possibly fractionally many) tokens of the word type “foo.” This goes by the name Laplace smoothing or add-δ smoothing, where δ is a parameter. We write:

$$P(w|l) = \frac{C_l(w) + \delta}{|V|\delta + \sum_{v \in V} C_l(v)}$$

where $l \in \{Sports, Business\}$, and $C_l(w)$ is the number of times the token w appears in news articles labeled l in the training set. As above, $|V|$ is the size of the total vocabulary we assume we will encounter (i.e., the dictionary size). Thus it forms a superset of the words used in the training and test sets. The value $|V|$ will be passed to the train method of your classifier as the argument `int v`. For this assignment, use the value $\delta = 0.00001$. With a little reflection, you will see that if we estimate our distributions in this way, we will have $\sum_{w \in V} P(w|l) = 1$. Use the equation above for $P(w|l)$ to calculate the conditional probabilities in your implementation.

Log Probabilities

The second gotcha that any implementation of a Naive Bayes classifier must contend with is underflow. Underflow can occur when we take the product of a number of very small floating-point values. Fortunately, there is a workaround. Recall that a Naive Bayes classifier computes

$$f(w) = \arg\max_{l} \left[P(l) \prod_{i=1}^{k} P(w_i|l) \right]$$

where *l* ∈ {*Sports*, *Business*} and w_i is the *i*th word token in a news article, numbered 1 to *k*. Because maximizing a formula is equivalent to maximizing the log value of that formula, *f*(*w*) computes the same class as

$$g(w) = \arg\max_{l} \left[\log P(l) + \sum_{i=1}^{k} \log P(w_i|l) \right]$$

What this means for you is that in your implementation you should compute the *g*(*w*) formulation of the function above rather than the *f*(*w*) formulation. Use the Java function `log(x)` which computes the natural logarithm of its input. This will result in code that avoids errors generated by multiplying very small numbers. Note, however, that your methods `p_l` and `p_w_given_l` should return the true probabilities themselves and ***NOT*** the logs of the probabilities.

This is what you should return in the `ClassifyResult` class: `log_prob_sports` is the value $\log P(l) + \sum_{i=1}^{k} \log P(w_i|l)$ with *l* = *Sports* and `log_prob_business` is the value with *l* = *Business*. The label returned in this class corresponds to the output of *g*(*w*). Break ties by classifying an instance as *Sports*.

Testing

We will test your program on multiple training and testing sets, and the format of testing commands will be:

```
java NewsClassifier <modeFlag> <trainFilename> <testFilename>
```

where `trainingFilename` and `testFilename` are the names of the training and testing dataset files, respectively. `modeFlag` is an integer from 0 to 3, controlling what the program will output:

- **0.** Prints the number of documents for each label in the training set
- **1.** Prints the number of words for each label in the training set
- **2.** For each instance in test set, prints a line displaying the predicted class and the log probabilities for both classes
- **3.** Prints the confusion matrix

In order to facilitate debugging, we are providing a sample input file and its corresponding output file for each mode. They are called `train.bbc.txt` and `test.bbc.txt` in the zip file. So, here is an example command:

```
java NewsClassifier <modeFlag> train.bbc.txt test.bbc.txt
```

As part of our testing process, we will unzip the file you submit, remove any class files, call `javac *.java` to compile your code, and then call the main method `NewsClassifier` with parameters of our choosing. Make sure your code runs on the computers in the department because we will conduct our tests on these computers.