Problem 3. A Naïve Bayes Classifier for Sentiment Analysis [65 points]

One application of Naïve Bayes classifiers is **sentiment analysis**, which is a sub-field of AI that extracts affective states and subjective information from text. One common use of sentiment analysis is to determine if a text document expresses negative or positive feelings. In this homework you are to implement a Naïve Bayes classifier for categorizing movie reviews as either **POSITIVE** or **NEGATIVE**. The dataset provided consists of online movie reviews derived from an IMDb dataset: https://ai.stanford.edu/~amaas/data/sentiment/ that have been labeled based on the review scores. A negative review has a score ≤ 4 out of 10, and a positive review has a score ≥ 7 out of 10. We have done some preprocessing on the original dataset to remove some noisy features. Each row in the training set and test set files contains one review, where the first word in each line is the class label (1 represents POSITIVE and 0 represents NEGATIVE) and the remainder of the line is the review text.

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 need to focus on is the implementation in the file `NaiveBayesClassifier.java` If you open that file, you will see the following methods that you must implement:

- `Map<Label, Integer> getDocumentsCountPerLabel (List<Instance> trainData)`
  This method counts the number of reviews per class label in the training set and returns a map that stores the (label, number of documents) key-value pair.
- `Map<Label, Integer> getWordsCountPerLabel(List<Instance> trainData)`
  This method counts the number of words per label in the training set and returns a map that stores the (label, number of words) key-value pair.
- `void train(List<Instance> trainData, int v)`
  This method trains 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.)
- `ClassifyResult classify(List<String> words)`
  This method returns the classification result for a single movie review.

In addition, you will need to implement a *k*-fold cross validation function as defined in the `CrossValidation.java`:

- `double kFoldScore(Classifier clf, List<Instance> trainData, int k, int v)`
  This method takes a classifier `clf` and performs `k`-fold cross validation on `trainData`. The output is the average of the scores computed on each fold. You can assume that the number of instances in `trainData` is always divisible by `k`, so the size

of each fold will the same. We will use the accuracy (number of correctly predicted instances / number of total instances) as the score metric.

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 movie review as a list of strings:

```
public class Instance {
        public Label label;
        public List<String> words;
}
```

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

```
public enum Label {POSITIVE, NEGATIVE}
```

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

```
public class ClassifyResult {
         public Label label;
         public Map<Label, Double> logProbPerLabel;
}
```

The *only* provided files you need edit are `NaiveBayesClassifier.java` and `CrossValidation.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 review 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 type 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: $P(w|Positive)$ and $P(w|Negative)$.

One might consider estimating the value of $P(w|Positive)$ by simply counting the number of tokens of type *w* and dividing by the total number of word tokens in all reviews in the training set labeled as *Positive*, but this method is not good enough in general because of the "unseen event problem," i.e., the possible presence of words 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 $P(foo|Positive)$ and $P(foo|Negative)$? 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.

What we do to get around this is 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. The conditional probability then is defined as:

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

where $l \in \{Positive, Negative\}$, and $C_l(w)$ is the number of times the tokens of word type *w* appears in reviews 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 **δ = 1**. With a little reflection, you will see that if we estimate our distributions in this way, we will have $\sum_{w \in V} P(w \mid l) = 1$. 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 Naïve 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 Naïve Bayes classifier computes

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

where $l \in \{Positive, Negative\}$ and w_i is the i^{th} word token in a review, 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

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

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.

This is what you should return in the `ClassifyResult` class: `logProbPerLabel.get(Label.POSITIVE)` is the value $\log P(l) + \sum_{i=1}^{k} \log P(w_i \mid l)$ with *l* = *Positive* and `logProbPerLabel.get(Label.NEGATIVE)` with *l* = *Negative*. The label returned in this class corresponds to the output of *g*(*w*). Break ties by classifying a review as *Positive*.

Testing

We will test your program on multiple training and test sets, using the following command line format:

```
java SentimentAnalysis <mode> <trainFilename> [<testFilename> | <K>]
```

where `trainingFilename` and `testFilename` are the names of the training set and test set files, respectively. `mode` is an integer from 0 to 3, controlling what the program will output. When mode is 0 or 1, there are only two arguments, `mode` and `trainFilename`; when the mode is 2 the third argument is `testFilename`; when mode is 3, the third argument is `K`, the number of folds used for cross validation. The output for these four modes should be:

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 accuracy score for `K`-fold cross validation

In order to facilitate debugging, we are providing sample training set and test set files called `train.txt` and `test.txt` in the zip file. In addition, we are providing the correct output for each of the four modes based on these two datasets, in the files `mode0.txt`, …, `mode3.txt`

For example, the command

```
java SentimentAnalysis 2 train.txt test.txt
```

should train the classifier using the data in `train.txt` and print the predicted class for every review in `test.txt`

The command

```
java SentimentAnalysis 3 train.txt 5
```

should perform **5**-fold cross-validation on `train.txt`

You are *not* responsible for handling parsing of the training and test sets, creating the classifier, or printing the results on the console. We have written the main class `SentimentAnalysis` for you, which will load the data and pass it to the method you are implementing. Do **NOT** modify any IO code.

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 `SentimentAnalysis` with parameters of our choosing. Make sure your code runs and terminates in less than 20 seconds for each provided test case on the computers in the department because we will conduct our tests on these computers.

Deliverables

Hand in your modified versions of `NaiveBayesClassifier.java` and `CrossValidation.java`. Also submit any additional helper `.java` files required to run your program (**including the provided ones in the skeleton**). Do not submit any other files including the data files (i.e., no `.class` or `.txt` files). All the `.java` files should be zipped as `<wiscNetID>-HW5-P3.zip` for submission to Canvas.