Assignment Rules

Making the assignment

You should make this assignment **individually**. You are allowed and encouraged to discuss ideas necessary to solve the assignment with your peers, but you should write the code by yourself. We will employ fraud checking software specifically created for programming classes such as this.
In case we suspect fraud, we can invite you to explain your code. Due to university policy cases of (suspected) fraud must be reported to the exam committee. The exam committee will then decide on possible disciplinary actions.
The deadline for this assignment is **Thursday, 17th of September 2020, 23:00, CEST**.

Grade and Scores

The grade of your assignment will depend on two important criteria:

1. Correctness: does your code work according to the specification in the assignment?
2. Code Quality and Style: is your code well written and understandable?

This assignment graded via **Codegrade** and counts for 20% towards your final grade. Note that this automated grade is not final: there is absolutely no guarantee that the automated grade provided by Codegrade will be your actual grade, as your work will also be assisted by human review. The weights for different elements of the assignment are:

Part of Assignment	Points
WordCounter Utility	1
WordCounter Probility	2
NaiveBayes Training and Classification	3
NaiveBayes using Files	1
NaiveBayes with ConfusionMatrix	1
Code Quality and Style	2
Total	10

When we ask you to implement a function using a **specific** algorithm, you will get no points if you use a different algorithm.

Handing in your assignment

- Any scores generated by **Codegrade** only give an indication of your possible grade. Final grades will only be given after your code has been checked manually, and may differ from the preliminary automated feedback provided. In some cases there may be additional testcases that are only executed after you hand in your final code.
- Deliberate attempts to trick **Codegrade** into accepting answers that are clearly not a solution to the assignments will be considered *fraud*.

Use the default package

Codegrade does not support `.java` files that are not in the default package. If the first line of your file contains `package somepackagename;`, it is not in the default package and this is not supported by our autograding tool. In Eclipse, you can move a class to the default package with the menu option *Move* in the *Refactor* menu.

Use no diacritical symbols or use UTF-8

By default, most Windows computers running Eclipse use the `latin1` character set to store text files, and as a consquence Eclipse does this as well. In this encoding, you can not use diacritical symbols, such as é or è. If you still want to use them, you should configure Eclipse to use UTF-8. This can be done by going to *Preferences* in the *Window* menu. Expand the *General* category and select *Wordspace*. In this settings tab, you should select *Other* for the *Text file encoding* and pick `UTF-8` in the choice box.

Code Quality and Style

Part of your grade will also depend on the quality and style of your code. **Codegrade** will check a number of rules automatically and compute a tentative score based on these rules. **Note that in the actual assignments, this score will likely be adjusted based upon manual inspection by human graders**. The automated tools do not find all issues, in and rare cases the automated graders may be too strict.
The rules applied by the automated grader come in four categories: *mandatory rules*, *important rules*, *slopy code rules* and *layout, style and naming rules*. This score is only an indication and is computed as follows:

- If any *mandatory rules* are broken, you get no points for code quality and style.
- If no *important rules* and *sloppy code rules* are broken, you get 1.5 points. For each *important rule* you break, 0.2 points are deducted. For each *sloppy code rule* you break, 0.1 points are deducted. It doesn't matter how often a particular rule is broken.
- If no *layout, style and naming rules* are broken, you get 0.5 points. For the first three violations, no points are deducted, but for each violation beyond that 0.1 point is deducted.
- Note that this score is only an indication of your code quality. Manual grading will always overrule the score indicated by Codegrade!

The detailed rules are explained in the "*Code quality and style guide*" of this course. They can be found on `https://erasmusuniversityautolab.github.io/FEB22012-StyleGuide/`

General Remarks

- The lecture slides usually contain a number of useful hints and examples for these assignments. If you get stuck, always check if something useful was discussed during the lectures. If you have question, feel free to contact us by e-mail.
- You can write your Java code with any editor your like. In the computer labs you will have access to BlueJ and Eclipse. We encourage you to use a good IDE, such as Eclipse, Netbeans or IntelliJ, as those have better features to help you with simple tasks and warn you in case of possible mistakes.

Naive Bayes Classifier

In the assignment of this week we will work with Machine Learning. Our task is to construct a Naive Bayes Classifier. These types of classifiers can be used to identify which class a certain document belongs to by making use of Bayesian statistics. For those who want to gain more insight into the mathematics behind Naive Bayesian Classifiers, there is a more in-depth introduction at the end of the document.

We will use our Bayesian classifier to differentiate between documents which are spam and which are not spam. In order to do this we first show our classifier many already classified documents, in order to *train* it. After this is done, we can proceed to identify unclassified documents as being spam or not spam. Which classification a document is given, depends on two factors.

- The fraction of already classified documents which were spam
- The frequency that certain words appear in documents which are as spam or not spam.

In order to keep track of these statistics, we first write a `WordCounter` class, which has the task of counting the number of times a certain word appears, as well as keeping track if the document which is being read is spam or not. After that, we use a selection of these `WordCounters` to create the `NaiveBayes` class. This class will first be fed a list of already classified documents, handled by the `WordCounters`. After this is done, the class can then be used to identify unclassified documents.

Using the methods used for training and classification, more advanced methods can also be constructed. For example, it might be useful to write a method which identifies multiple unclassified documents, and writes the classifications into another document. Furthermore, we also construct a `ConfusionMatrix` class, which is used to give information about the power of the classifier.

Data format

To keep the data format simple, documents are always considered to be a single line of words without interpunction contained in a `String`. If a document has a classification, the first character in the String is this classification. If the document is considered spam, it is 1. If it is considered no spam, it is 0. An example of a five word document that is considered spam, is:

```
1 money money money free free
```

Sometimes, it is useful to have multiple documents in a single file. In that case, every line is considered a document. Thus, a file containing 5 documents, of which three have the classification spam, could look as follows:

```
1 money money money free free
0 good good good money
1 free money money good
0 free good good
1 good free money
```

Such a file can be used to train a Naive Bayes Classifier. A trained classifier can then be used to determine the probability that a new document with a classification is spam or not. A document without a classification looks very similar to a document with a classification: only the label at the start of the document is missing. An example of a document without classification is:

```
money good money free free
```

Utility Methods

In the first part of the assignment you will create the `WordCounter` class, which main purpose is to count how many times a certain word appears in the documents it has seen. Furthermore, some utility methods have to be written in order to extract the necessary information from an input file. The `WordCounter` class must have the following methods (you are allowed to add more if this is helpful to you):

```
1 public WordCounter(String focusWord) {...}
2 public String getFocusWord() {...}
3 public void addSample(String document) {...}
```

The constructor of the `WordCounter` class takes in a `String focusWord`, which is the word that is be counted by the current `WordCounter` object. Furthermore, we should be able to ask a `WordCounter` object which focus word was passed to its constructor. The `getFocusWord()` method should return the focus word when it is invoked.
Second, the `addSample(String document)` method must be able to process a document, extract all useful information from it and update the information in the current object. The introduction of this assignment has a description of what a document looks like. You can assume the `document` always starts with a `0` or `1` and contains lower case words separated by spaces.
The `addSample(...)` method should extract the following statistics from the String:

- Whether the `document` in question has true classification spam or no spam.
- How many times the `focusWord` appears in the document.
- The total number of words that appear in the document.

By utilizing this information, an instance of the `class` should keep track of the following statistics:

- The total number of words that appear in all the documents considered no spam (0).
- The total number of words that appear in all the documents considered spam (1).
- How many times the `focusWord` appears in all the documents considered no spam (0).
- How many times the `focusWord` appears in all the documents considered spam (1).

Example: if a `WordCounter` with focus word `good` processes three documents the following way:

```
1 WordCounter wc = new WordCounter("good");
2 System.out.println(wc.getFocusWord());
3 wc.addSample("1 good bad bad bad");
4 wc.addSample("0 bad good good");
5 wc.addSample("0 bad good");
```

After running this code, it should print "`good`" and the following should hold:

- The total number of words in no spam documents is 5
- The total number of words in spam documents is 4
- The number of times the focus word appears in no spam documents is 3
- The number of times the focus word appears in spam documents is 1

Note that in order to inspect the state of the `wc` object, you should either implement a method that can perform this for you, use a *debugger* or add a temporary `System.out.println` statement to the `addSample` method (but do not forget to remove it when you hand in your code!.

Probability Methods

Now that the utility Methods are in place, you must implement some methods that compute conditional and unconditional probabilities that a document is spam. If you are interested in the mathematics behind this, you can read the *Introduction to Bayes Classifiers* at the end of this document, but this is not required. The new methods are to be added to the `WordCounter` class. Afterwards, the `WordCounter` class should have at least the following public methods (you are allowed to add more):

```
1 public WordCounter(String focusWord) {...}
2 public String getFocusWord() {...}
3 public void addSample(String document) {...}
4
5 public boolean isCounterTrained() {...}
6 public double getConditionalNoSpam() {...}
7 public double getConditionalSpam() {...}
```

First of all the `isCounterTrained()` method can be used to check if a `WordCounter` has seen enough information to consider it properly trained. We consider that a `WordCounter` is *trained* if and only if all three of the folllowing hold: 1) it has seen its `focusWord` in at least one document, 2) it has seen at least one spam document and 3) it has seen at least one no spam document. If this is the case, `isCounterTrained()` should return `true` and it should return `false` otherwise.

The methods `getConditionalNoSpam()` and `getConditionalSpam()` should return estimates for the probability that a random word from a document is the `focusWord`, conditional on the fact that the document is classified as respectively no spam or spam. In practice these can be calculated using the following ratios:

$$\texttt{getConditionalNoSpam()} \text{ returns } \frac{\#\ \texttt{focusWord} \text{ over all not spam}}{\#\ \text{words over all not spam}}$$

$$\texttt{getConditionalSpam()} \text{ returns } \frac{\#\ \texttt{focusWord} \text{ over all spam}}{\#\ \text{words over all spam}}$$

Lastly, if `getConditionalNoSpam()` or `getConditionalSpam()` are called before the `WordCounter` has been *trained*, an `IllegalStateException` should be thrown.

Example: Suppose that we execute the following code after the example for the utility methods:

```
1 System.out.println(wc.getConditionalSpam());
2 System.out.println(wc.getConditionalNoSpam());
```

then the expected output is:

```
0.25
0.6
```

Training and Classification Methods

With the methods necessary to calculate the individual conditional probabilities for every word, we can now construct the Naive Bayesian Classifier. You need to create a `NaiveBayes` class which should contain at least the following public methods (you can add more if you want):

```
1 public NaiveBayes(String[] focusWords) {...}
2 public void addSample(String document) {...}
3 public boolean classify(String unclassifiedDocument) {...}
```

The constructor of the `NaiveBayes` class has a `String[] focusWords` variable as input, which is an array containing the focus words. Using this array, the constructor should initialize a `private WordCounter` array, where every `WordCounter` corresponds to a `String` from `focusWords`.
When the method `addSample` is called for a document with a classification, the provided document should be processed by all the `WordCounter` objects managed by the `NaiveBayes` object. Furthermore, the `NaiveBayes` object should keep track of the following information:

- The total number of documents considered as no spam (0) it has seen.
- The total number of documents considered as spam (1) it has seen.

The can be used to compute the initial scores that a document is spam $P(\text{spam})$ and that the document is no spam $P(\neg\text{spam})$ using the formula:

$$P(\text{spam}) = \frac{\#\text{ spam documents seen}}{\#\text{ total documents seen}} \tag{1}$$

$$P(\neg\text{spam}) = \frac{\#\text{ no spam documents seen}}{\#\text{ total documents seen}} \tag{2}$$

Lastly, the `classify(String unclassifiedDocument)` takes in a document as `String` without a classification. It should then determine if it the probability that this document is spam is greater than the probability that is it not spam. This can be done using the following steps:

1. Compute an initial *spam score* and a *no spam score* using Equations 1 and 2.
2. For each `word` in `unclassifiedDocument`, check if this `NaiveBayes` object holds a `WordCounter` that has `word` as its focus word. If such a `WordCounter` exists, use it to:
   (a) Set the *spam score* to the old *spam score* multiplied by `getConditionalSpam()`.
   (b) Set the *no spam score* to the old *no spam score* multiplied by `getConditionalNoSpam()`.
3. Determine if the *no spam score* is smaller than the *spam score*.

Example: Suppose the following code is executed:

```
1  String [] words = {"good", "bad"};
2  NaiveBayes nb = new NaiveBayes(words);
3  nb.addSample("1 good bad bad bad casino");
4  nb.addSample("0 bad good good pizza");
5  nb.addSample("0 bad good tapas");
6  System.out.println(nb.classify("good"));
7  System.out.println(nb.classify("bad"));
8  System.out.println(nb.classify("good bad bad"));
9  System.out.println(nb.classify("pizza"));
10 System.out.println(nb.classify("casino"));
```

it should print the lines `false`, `true`, `true`, `false` and `false`.

Training and Classification using Files

Now that we can train a `NaiveBayes` object using sample documents, and use a trained object to `classify` new documents, we should expand the functionality of the `NaiveBayes` class to perform these tasks on files. We will add two methods to the `NaiveBayes` class, after at which it should at least contain the following public methods:

```
1 public NaiveBayes(String[] focusWords) {...}
2 public void addSample(String document) {...}
3 public boolean classify(String unclassifiedDocument) {...}
4
5 public void trainClassifier(File trainingFile) throws IOException {...}
6 public void classifyFile(File input, File output) throws IOException {...}
```

When `trainClassifier(File file)` is executed, the `File trainingFile` passed as an argument should be read in such a way that every line of the file is considered a document with a classification that should be processed to train the `NaiveBayes` object. The effect should be the same as feeding every line of the file to the `addSample` method of the `NaiveBayes` object `trainClassifier` is called on. If the file can not be read, or does not exist, the method should throw either an `IOException` or a `FileNotFoundException` (it is also possible to let this Exception be thrown by the object you use for reading the file).
The `classifyFile(File input, File output)` method is intended to classify multiple documents using one method. The parameter `input` denotes a file which contains a document without classification on each line. The parameter `output` denotes a file where the classification results should be written to: either a `1` if the corresponding line in the `input` file is classified as spam, and a `0` if it is classified as not span.
Example: suppose that a file `newdata.txt` exists with the following contents:

```
good
bad
good bad bad
pizza
casino
```

Now if the following piece of code is executed

```
1 String [] words = {"good", "bad"};
2 NaiveBayes nb = new NaiveBayes(words);
3 nb.trainClassifier(new File("traindata.txt"));
4 nb.classifyFile(new File("newdata.txt"), new File("classifications.txt"));
```

then afterwards the file `classifications.txt` should contain

```
0
1
1
0
0
```

Measuring Classification Accuracy

Now that it is possible to train a classifier based on a data set, the final step is to compute how good our classifier performs. In machine learning, this is often done by splitting your training data into a *training* set and a *test* set. The training data is then used to train the classifier, and the predictions made by the classifier are then compared against the actual classifications of the documents in the test set. This way, it is possible to compute four *accuracy* measures:

true positives are cases where the classifier predicts 1 for a document that has classification 1

false positives are cases where the classifier predicts 1 for a document that has classification 0

true negatives are cases where the classifier predicts 0 for a document that has classification 0

false negatives are cases where the classifier predicts 0 for a document that has classification 1

Introduce a new class `ConfusionMatrix` that has at the least the following four public methods (more are allowed):

```
1 public int getTrueNegatives() {...}
2 public int getTruePositives() {...}
3 public int getFalseNegatives() {...}
4 public int getFalsePositives() {...}
```

Now we add one more method to the `NaiveBayes` class, so that it has the following set of methods:

```
1 public NaiveBayes(String[] focusWords) {...}
2 public void addSample(String document) {...}
3 public boolean classify(String unclassifiedDocument) {...}
4
5 public void trainClassifier(File trainingFile) throws IOException {...}
6 public void classifyFile(File input, File output) throws IOException {...}
7
8 public ConfusionMatrix computeAccuracy(File testdata) throws IOException {...}
```

The argument `test` refers to a file with one each line a document including classification, i.e. the same structure as the file that can be fed to the `trainClassifier` method. The result of this method should be an object of type `ConfusionMatrix`, from which the four accuracy measures as they occur in the `testdata` file can be obtained.
Suppose the files `traindata.txt` and `textdata.txt` contain the data on the next page. Now after the following piece is executed

```
1 String [] words = {"good", "bad"};
2 NaiveBayes nb = new NaiveBayes(words);
3 nb.trainClassifier(new File("traindata.txt"));
4 ConfusionMatrix cm = nb.computeAccuracy(new File("testdata.txt"));
5 System.out.println(cm.getTruePositives());
6 System.out.println(cm.getFalsePositives());
7 System.out.println(cm.getTrueNegatives());
8 System.out.println(cm.getFalseNegatives());
```

The following should be printed:

```
3
2
4
1
```

Contents of data files

traindata.txt	testdata.txt
1 good bad bad bad casino 0 bad good good pizza 0 bad good tapas	1 bad bad casino 1 bad pizza 1 bad 1 good good 0 good good pizza 0 good tapas 0 good 0 good casino 0 bad pizza bad casino 0 bad

Background: Introduction to Bayes Classifiers

Note: this is a short introduction to Bayes Classifiers. You do not have to read this or understand this to be able to make the assignment!

Let's say that you've been tasked with identifying spam in a list of emails. While it might not be very difficult to do manually, one could imagine that this task could get bothersome. Therefore, after classifying a number of documents, you may consider if there exists some kind of underlying pattern in the spam. Let's say that you classified the following emails, which may look something like this (after removing punctuation marks):

- 0 hey how are you doing
- 1 hey free money click here
- 0 do you want free smartphone

Here 0 means that the email is classified as not being spam, while 1 means that the email is classified as spam. Maybe you notice that the word "free" is more common in emails which are known to be spam than emails that are known to not be spam. Furthermore, you see that words as "hey" or "you" are common in both types of emails. As the wise econometrician that you are, you attempt to quantify this idea using Bayesian statistics.
Consider an unclassified document consisting of n words each, such that the random variables $X_1, \dots, X_n$ represent the n words in the document (for example, X_3 is the random variable which generates the 3th word in the document). Also, let us assume that the respective realizations of $X_1, \dots, X_n$, given as $x_1, \dots, x_n$ can take values from the word set $W = \{w_1, \dots, w_k\}$. In the example given above, we have $x_1 =$ "hey", $x_2 =$ "how", $x_3 =$ "are", $x_4 =$ "you", $x_5 =$ "doing". Furthermore, define the random variable S with realization s as follows:

$$S = \begin{cases} 0 & \text{if the document is spam} \\ 1 & \text{if the document is not spam} \end{cases}$$

If we ask ourselves how we should decide upon the classification of a document, we might conclude to classify a document as spam if the conditional probability that a document is spam is larger than the probability that a document is not spam. In mathematical terms, this corresponds to the following statement:

$$P(S = 1 | X_1 = x_1, \dots, X_n = x_n) > P(S = 0 | X_1 = x_1, \dots, X_n = x_n)$$

Applying Bayes theorem and multiplying with $P(X_1 = x_1, \dots, X_n = x_n)$ gives us the following equivalent expression:

$$P(S = 1)\frac{P(X_1 = x_1, \dots, X_n = x_n | S = 1)}{P(X_1 = x_1, \dots, X_n = x_n)} > P(S = 0)\frac{P(X_1 = x_1, \dots, X_n = x_n | S = 0)}{P(X_1 = x_1, \dots, X_n = x_n)}$$

$$\Updownarrow$$

$$P(S = 1)P(X_1 = x_1, \dots, X_n = x_n | S = 1) > P(S = 0)P(X_1 = x_1, \dots, X_n = x_n | S = 0)$$

In order to make the problem more accessible, Naive Bayes Classifiers assume that $X_1, \dots, X_n$ are generated identically and independent from each other. Therefore, the expression above reduces to the following:

$$P(S = 1)\prod_{i=1}^{n} P(X = x_i | S = 1) > P(S = 0)\prod_{i=1}^{n} P(X = x_i | S = 0)$$

Now all that is left to do is find estimates for the unconditional probabilities $P(S = 0)$ and $P(S = 1)$, as well as the conditional probabilities $P(X = w_i | S = 0)$ and $P(X = w_i | S = 1)$, with

$i \in \{1, \ldots, k\}$. Assuming that we have multiple already classified documents, these are estimated as follows:

$$\widehat{P}(S = 1) = 1 - \widehat{P}(S = 0) = \frac{\text{total \# of spam documents}}{\text{total \# of documents}}$$
$$\widehat{P}(X = w_i | S = 0) = \frac{\text{\# } w_i \text{ over all not spam}}{\text{\# words over all not spam}}$$
$$\widehat{P}(X = w_i | S = 1) = \frac{\text{\# } w_i \text{ over all spam}}{\text{\# words over all spam}}$$

Example estimation and classification

Let's consider an example where we have to estimate the unconditional and conditional probabilities, as well as classify an unknown document. The data on which we train our network is given below.

- 1 money money money free free
- 0 good good good money
- 1 free money money good
- 0 free good good
- 1 good free money

Using the formulas specified above, we find the following estimated probabilities:

$$\widehat{P}(S = 0) = \frac{2}{5}$$
$$\widehat{P}(S = 1) = \frac{3}{5}$$

$$\widehat{P}(X = \text{"money"} | S = 0) = \frac{1}{7}$$
$$\widehat{P}(X = \text{"free"} | S = 0) = \frac{1}{7}$$
$$\widehat{P}(X = \text{"good"} | S = 0) = \frac{5}{7}$$
$$\widehat{P}(X = \text{"money"} | S = 1) = \frac{6}{12} = \frac{1}{2}$$
$$\widehat{P}(X = \text{"free"} | S = 1) = \frac{4}{12} = \frac{1}{3}$$
$$\widehat{P}(X = \text{"good"} | S = 1) = \frac{2}{12} = \frac{1}{6}$$

Now let's say that we want to classify the following document:

- good good money

Using the probabilities that were found above, we find the following:

$$P(S=1)\prod_{i=1}^{n} P(X=x_i|S=1) = \left(\frac{3}{5}\right)\left(\frac{1}{6}\right)^2\left(\frac{1}{2}\right) \approx 0.0083$$

$$P(S=0)\prod_{i=1}^{n} P(X=x_i|S=0) = \left(\frac{2}{5}\right)\left(\frac{5}{7}\right)^2\left(\frac{1}{7}\right) \approx 0.0292$$

In this case, because $0.0292 \geq 0.0083$, we would classify the document as not spam.