COMPUTER SCIENCE 21A (SPRING, 2020) DATA STRUCTURES AND ALGORITHMS

PROGRAMMING ASSIGNMENT 3 – SPAM OR SAFE?

Overview:

In this assignment, you will be implementing a simple Bayesian email spam filter. In general, the filter will receive and store emails and then filter out any emails that are above a certain spam threshold. To filter out these emails, the filter will need to be "trained" with provided emails that have been declared by the user as spam or safe. The filter will be represented as a Java class called `SpamFilter` that has the following capabilities:

- Calculate the "spamicity" of a word. The "spamicity" of a word is the probability that an email containing the word is a spam email. In order to calculate the spamicity of a word, you will need to track four items:

  1. How many spam emails has the filter seen in training data.
  2. The number of spam emails the word has appeared in. In order to retrieve this information efficiently, you will be using a hash table.
  3. How many safe emails has the filter seen in training data.
  4. The number of safe emails the word has appeared in. In order to retrieve this information efficiently, you will be using a hash table.

  More details on how to implement this functionality will be discussed later, but for now think of any additional fields you may need in the `SpamFilter` class and how you will be using a hash table to implement items 2 and 4 above.

- Calculate the "spam score" of an email using the spamicity of each word in the email. This will be done using a provided formula which you can find on page 8.

- Receive a collection of emails. You can think of this mimicking an email inbox when a bunch of emails arrive. However, this spam filter needs to manually filter out the spam, it is not done automatically as in a browser email service like Gmail.

- Filter out all spam emails above a certain threshold "spam score". To both store received emails and efficiently remove all those above a specified threshold, you will be using a modified maximum priority queue. This queue will store emails in a heap as discussed in class and will compare emails based on their spam score.

- Train the filter given a collection of emails that have been marked by the user as spam or safe. When implementing this functionality, you will need to consider which fields of `SpamFilter` will need to be updated. More details on the implementation of this functionality will be discussed on page 9.

- Consider the scenario where your filter has stored some emails and has yet to filter them. However, some training data arrives that indicates some stored emails are now above the filter's spam score threshold and that some others are now below the filter's spam score threshold. For this reason, your filter will have to be able to efficiently update emails' spam scores for future filtering. Furthermore, your filter can be trained multiple times.

Design Strategy:

The implementation of the filter will require some time, but for now hopefully you have a general idea of the behavior of the filter and some idea of how you may implement it. The overview is meant to give you an idea of the final goal of this assignment, but we recommend that you start by implementing the data structures you will be using to implement the `SpamFilter` class.

Step 1: Implement *and test* the `HashTable` class (implementation details on pages 4 and 5). This will first require you to implement the `Entry` class (implementation details on page 4).

You will be using the hash table throughout the assignment so make sure you give yourself enough time to implement this class properly using your knowledge from lecture. Make sure to test it thoroughly using JUnit tests.

Make sure to consider these questions in the process of writing this class:

- How will you implement the table's hash function?
- What if table's load factor is surpassed? What is involved in resizing a hash table?

Step 2: Implement the `Email` class (implementation details on page 5). This class' sole purpose is to represent an email that will be stored in the `PriorityQueue` and in `SpamFilter`. The calculations required for an email's spam score will be done in `SpamFilter`. For now, just consider an `Email` as an item stored in the `PriorityQueue` with its spam score being its priority.

Step 3: Implement *and test* the `PriorityQueue` class (implementation details on pages 5 through 7). This class is a non-generic data structure that is meant solely for storing emails for the `SpamFilter`

This class will serve as one of the crucial components of the `SpamFilter` class so make sure you give yourself enough time to implement this class properly using your knowledge from lecture. Make sure to test it thoroughly using JUnit tests.

Make sure to consider these questions in the process of writing this class:

- As mentioned at the top of this page, you will have to be updating the spam scores of emails when new training data arrives. Therefore, the `PriorityQueue` class has an `updateSpamScore()` method that updates an email to have a new spam score. How can you implement it so that its runtime is O(log(n))?

  Hint: What data structure could you use internally in your `PriorityQueue` besides a heap that allows you to quickly identify which heap element needs to be updated and "heapified"? Will you need to change other methods in this class based on your decision?

- One of the methods of the `PriorityQueue` class returns a `String` array that contains a descending ordering of the emails stored in the `PriorityQueue` based on their spam score. While it is true that the queue's internal heap allows efficient retrieval of the email with the maximum spam score (i.e. priority), it does not store its elements from highest to lowest. Therefore, you will need to sort a copy of the heap and return the required information of the method (`rankEmails()`, on page 6). Which sorting algorithm covered in class would be appropriate to implement this functionality?

  Hint: A heap is a partially sorted array since emails with higher spam scores will be located at the front of the array and lower spam score emails are located towards the rear.

Step 4: Implement the `SpamFilter` class (implementation details on pages 7 through 10). Read the implementation details of this carefully as we have provided you with some of the logic you will need to implement this class correctly. We also recommend that you look at the provided JUnit test file for this class which gives example calculations for spamicity and spam score. The test file also contains simple tests of some of the other methods in `SpamFilter`

Implementation Details:

Below is the list of the classes and methods you need to implement. **You must provide implementations of all the methods below matching these exact signatures.** For `PriorityQueue` you are also required to use a specific field. Besides this class, you are allowed to use any fields that you see fit.

You cannot use any Java data structures besides arrays, but feel free to implement additional structures that have been covered in lecture. Any non-test classes that you create should be placed into the `main` package.

You should not be importing any Java-provided libraries except `java.util.NoSuchElementException` in PriorityQueue.java.
This includes `java.util.Arrays.`

Entry.java

This class will be used in the implementation of the `HashTable` class as the data type of the internal table array. For the purposes of this assignment, your hash table will only have to store `Strings` as keys. Therefore, the `Entry` class will be storing a (key, value) pair of a `String` key and a value of generic type `V`.

`public Entry(String key, V value)` – this constructs an `Entry` with a provided (key, value) pair.

`public String getKey()` – gets the key stored in this `Entry`

`public V getValue()` – gets the value stored in this `Entry`

`public void setValue(V value)` – sets the value stored in this `Entry`

HashTable.java

This class provides the functionality of a generic hash table. Key values will always be `Strings` but values are of a generic type `V`. This class contains one instance field and one static field that you must use in your implementation. **Keep these fields** `public`**.**

`public Entry<V>[] entries` – you should be using this field as the internal array of your hash table. You **must use open addressing hashing**. You should not be using chaining of any kind.

`public static double LOAD_FACTOR` – you should be using this to determine when more space needs to be added to the hash table. Specifically, when the number of entries in the table divided by the capacity of `entries` surpasses the load factor, your hash table will need to be resized. **Do not make this field** `final`

`public HashTable()` – constructs an empty `HashTable`. You will find that one line has been written for you which allows you to construct a generic array of type `Entry<V>`. Feel free to change the initial array size, but do not change anything else about this line of code. *However, you may find this line useful elsewhere in this class.*

`public V get(String key)` – this should get the generic value stored in this `HashTable` that is associated with the given key or `null` if there is no entry in the `HashTable` with the provided key.

`public void put(String key, V value)` – this should create an entry in the `HashTable` that associates a given `String` key with a given generic value. If the given key already exists in this `HashTable` then you should overwrite the existing entry’s value with the provided value.

`public V delete(String key)` – this should delete the entry from this `HashTable` that has the provided key. This should return the generic value of the entry that was deleted or `null` if no entry in this `HashTable` has the given key.

`public String[] getKeys()` – this should return an array of the keys stored in this `HashTable`. The returned array can contain the keys in any order.

`public int size()` – returns the number of entries stored in this `HashTable`.

Email.java

This class represents an email. An `Email` has a unique identifier, an array of the words in the `Email`, and the `Email`'s "spam score". The details of calculating an `Email`'s spam score will be discussed later. For now, just treat it as a nonnegative integer.

`public Email(String id, String contents)` – constructs an `Email` object given a unique identifier and a `String` of the `Email`'s contents. The constructor should also initialize the `Email`'s spam score. You can set it to any negative integer.

`public String getID()` – returns the unique identifier of this `Email`.

`public String[] getWords()` – returns the words contained in this `Email`. You may assume that the words of an `Email` can be retrieved from the contents passed into the constructor as `String`s that are separated by a space " ".

`public int getSpamScore()` – returns the spam score of this `Email`.

`public void setSpamScore(int score)` – sets the spam score of this `Email`.

PriorityQueue.java

This class provides the functionality of a specialized heap-based maximum priority queue for storing `Email` objects. `Email` objects will be ordered in the internal heap based on their "spam score". Therefore, you can think of the "spam score" as the priority of an `Email`. This class is provided with the following field:

`public Email[] heap` – this is the internal maximum heap used in the implementation of the `PriorityQueue`. The heap is made up of `Email`s that will be compared based on their spam scores. **Keep this field** `public`**.**

`public PriorityQueue()` – constructs an empty `PriorityQueue`.

`public void insert(Email e)` – this should add an `Email` to the `PriorityQueue`. The internal heap should be re-adjusted based on `e`'s spam score.

`public void updateSpamScore(String eID, int score)` – this should update the spam score of the `Email` with the given ID. Following the update of the `Email`'s score, the `Email`'s location in the internal heap should be re-adjusted based on the `Email`'s new, updated spam score. If no `Email` with the given ID exists in this `PriorityQueue`, throw a `NoSuchElementException`.

`public int getMaximumSpamScore()` – this should return the spam score of the `Email` with the maximum spam score in this `PriorityQueue`. If the queue is empty, throw a `NoSuchElementException`.

`public Email extractMaximum()` – this should remove the `Email` from this queue with the highest spam score. This should return the `Email` that was removed. If there is no `Email` to be removed, throw a `NoSuchElementException`.

`public String[] getIDs()` – returns the identifiers of the `Emails` stored in this `PriorityQueue` in any order.

`public String[] rankEmails()` – returns the unique identifiers and spam scores of the `Email`s stored in this `PriorityQueue` sorted from highest to lowest spam score. Each element of the returned array should be a `String` where the identifier and spam score are separated by two spaces and dashes " -- ".

For instance, suppose this `PriorityQueue` contained `Email` objects with identifiers
`15a39fd4ebdd4140b384e1086b924424`,
`40f96c28d4a34a8d8657272c0a67ff9e`,
`8719d4cadb62480b97177d8f7fa71c96`,
and spam scores 1, 2, and 3 respectively.

This method should return the following array:

```
[    8719d4cadb62480b97177d8f7fa71c96 -- 3,
     40f96c28d4a34a8d8657272c0a67ff9e -- 2,
     15a39fd4ebdd4140b384e1086b924424 – 1    ]
```

`public String[] getWords(String id)` – returns the words of an `Email` stored in this `PriorityQueue` with the provided ID or `null` if there is no `Email` in this `PriorityQueue` with the provided ID.

`public int size()` – returns the number of entries stored in this queue.

Recommended Methods:
Below are some additional helper methods that we recommend including in your implementation of this class. These will allow you to condense your code for use in the implementation of the methods of the class that require re-adjustment of the queue's internal heap.

`private void swap(int i, int j)` – this swaps the `Email`s stored at indexes i and j in this `PriorityQueue`'s internal heap.

`private void heapifyDown(int index)` – this applies the heapify down algorithm from lecture on the `Email` stored at `index` in this `PriorityQueue`.

`private void heapifyUp(int index)` – this applies the heapify up algorithm from lecture on the `Email` stored at `index` in this `PriorityQueue`.

SpamFilter.java

This class provides the functionality of a simplified Bayesian email spam filter. A `SpamFilter` can be trained based on a set of `Email`s marked spam or not. Then, a `SpamFilter` can receive a set of `Email`s to be stored. Then, the stored `Email`s will be filtered based on the `Email`s' spam scores and the filter's threshold value.

`public SpamFilter(int threshold)` – constructs a `SpamFilter` with a threshold. The threshold will be used in the filtering process to determine which `Email`s will be filtered out. You can think of the threshold as the "strictness" of the filter. That is, by adjusting the threshold, certain `Email`s may or may not be filtered.

`public void setThreshold(int threshold)` – updates this `SpamFilter` to have a new threshold value. The threshold will be guaranteed to be a positive integer.

`public double getSpamicity(String word)` – returns the "spamicity" of a word. The spamicity of a word is the probability than an email is a spam given that it contains this word. This will be calculated using the following formula:

$$P(email\ is\ spam|word\ in\ email) = \frac{P(word\ in\ email|email\ is\ spam)}{P(word\ in\ email|email\ is\ spam) + P(word\ in\ email|email\ is\ safe)}$$

To do this, we will first need to calculate the probability that a spam email contains the given word. To do this, you will be utilizing the two of the items that your filter class needs to be tracking that were discussed in the overview on page 1:

- The total number of spam emails that have been seen in the training data (item 1).
- The number of spam emails a given word has appeared in (item 2).

$$P(word\ in\ email|email\ is\ spam) = \frac{\#\ spam\ training\ emails\ word\ appears\ in}{total\ \#\ of\ spam\ training\ emails}$$

We will also need to calculate the probability that a safe email contains the given word. To do this, you will be utilizing the two of the items that your filter class needs to be tracking that were discussed in the overview on page 1:

- The total number of safe emails that have been seen in the training data (item 3).
- The number of safe emails a given word has appeared in (item 4).

$$P(word\ in\ email|email\ is\ safe) = \frac{\#\ safe\ training\ emails\ word\ appears\ in}{total\ \#\ of\ safe\ training\ emails}$$

Note: If a word appears in a *received* email that has never appeared in any *training* emails, then it should have a spamicity of 0.

`public int calculateSpamScore(String[] words)` – returns the spam score of an array of words from an `Email`. The spam score will be calculated as follows:

```
spamScore ← 0
for each word in email do
   spamicity ← getSpamicity(word)
   if spamicity ≥ 0.9 then
      spamScore ← spamScore + 4
   else if spamicity ≥ 0.75 then
      spamScore ← spamScore + 2
   else if spamicity ≥ 0.5 then
      spamScore ← spamScore + 1
return spamScore
```

Note: For example calculations for `getSpamicity()` and `calculateSpamScore()`, please look at the provided JUnit test file `StudentFilterTest.java`.

`public void receive(Email[] emails)` – this should store the `Email`s in `emails` in this `SpamFilter`. You can think of this as a new set of `Email`s that have arrived in the "inbox" that will be filtered later.

`public String filter()` – this should remove all `Email`s that are stored in this `SpamFilter` that have spam scores greater than or equal to this `SpamFilter`'s threshold value. This must return the unique identifiers of the `Email`s that are removed on separate lines. If no `Email`s are removed, then return the empty String "".

For instance, suppose this `SpamFilter` filtered the `Email` objects with identifiers

```
15a39fd4ebdd4140b384e1086b924424,
40f96c28d4a34a8d8657272c0a67ff9e,
8719d4cadb62480b97177d8f7fa71c96
```

This method should return the following `String`:

```
8719d4cadb62480b97177d8f7fa71c96
40f96c28d4a34a8d8657272c0a67ff9e
15a39fd4ebdd4140b384e1086b924424
```

`public void train(Email[] emails, boolean[] isSpam)` – this passes this `SpamFilter` a set of `Email`s that have been labelled as spam or not. The goal of this method is twofold:

1. Update the number of spam or safe emails each word in each email in the `emails` parameter using the `isSpam` parameter.

2. Update the spam scores using this new training data of the Emails that are currently stored in the filter.

The logic you should be applying in this algorithm is as follows:

```
for i ← 0 to emails.length do
  if isSpam[i] then
     for each word w in emails[i] do
        add 1 to # of spam emails w is in if w is unmarked
        mark that # spam emails with w has been updated
     for each word w in emails[i] do
        unmark that # spam emails with w has been updated
     increase count of processed spam emails
  else
     for each word w in emails[i] do
        add 1 to # of safe emails w is in if w is unmarked
        mark that # safe emails with w has been updated
     for each word w in emails[i] do
        unmark that # safe emails with w has been updated
     increase count of processed safe emails

for each stored email e in the filter
  re-calculate and update spam score of e
```

You should use this logic when implementing this method, the one thing left for you to decide is how you will “mark” and “unmark” when the number of spam or safe emails a word has occurred in has been updated.

Note: you may assume that `emails` and `isSpam` are non-`null`, have the same length and do not have any `null` entries.

`public String getEmailRanking()` – this should return a `String` the unique identifier followed by its spam score of each `Email` stored in this `SpamFilter` sorted from highest to lowest based on spam score. The identifier and spam score should be separated by two dashes and two spaces " -- ". If the are no `Emails` stored in the `SpamFilter` then return the empty `String` "".

For instance, suppose this `SpamFilter` contained `Email` objects with identifiers `15a39fd4ebdd4140b384e1086b924424`, `40f96c28d4a34a8d8657272c0a67ff9e`, `8719d4cadb62480b97177d8f7fa71c96`, and spam scores 1, 2, and 3 respectively.

This method should return the following `String`:

```
8719d4cadb62480b97177d8f7fa71c96 -- 3
40f96c28d4a34a8d8657272c0a67ff9e -- 2
15a39fd4ebdd4140b384e1086b924424 -- 1
```

JUnit Tests

You must write JUnit tests for `HashTable.java` **and** `PriorityQueue.java`. Please write your JUnit tests in the provided files `StudentTableTest.java` **and** `StudentQueueTest.java` located in the `tests` package.

You have also been provided with `ExampleFilterTest.java`. This file **should not be edited.** It will help you ensure that you are using the exact Strings that are needed to implement various methods. **If you don't pass these tests, you will most likely fail the grading tests.**

You will also find an additional file used by the provided JUnit tests in the `tests` package called `EmailFileReader.java`. This file contains utility methods that can be used for reading training and receiving data from data files. Three of these data files that are used by the provided tests can be found in the zip on LATTE (`student_receive.txt`, `student_training_big.txt`, and `student_training_small.txt`). **You should not edit any of these files.**

Submission Details

- For every non-test method you write, you must provide a **proper JavaDoc** comment using `@param`, `@return`, etc. that also **includes the method's runtime.**
- At the top of **every file** that you submit (including tests) please leave a comment with the following format. Make sure to include **each of these 6 items.**

```
/**
 * <A description of the class>
 * Known Bugs: <Explanation of known bugs or "None">
 *
 * @author Firstname Lastname
 * <your Brandeis email>
 * <Month Date, Year>
 * COSI 21A PA3
 */

>Start of your class here<
```

- You must use the skeleton Eclipse project.
- It must be named **LastnameFirstname-PA3.** (Use Eclipse's Refactor > Rename)
- Submit a .zip file of your Eclipse project named **LastnameFirstname-PA3.zip**
- Late submissions will not receive credit.
- **Do not use any Java provided data structures other than arrays.**
- **Do not import any Java libraries besides** `java.util.NoSuchElementException`

Authors

This assignment was written by Chami Lamelas.