Homework #1: Cosines o' the Times

Due Thursday, September 8 at 11:59 p.m.

In this assignment you will use cosine similarity to detect similarities between web pages. The goals of this assignment are to familiarize you with our course infrastructure, let you practice object-oriented programming with Java, and introduce you to the collections framework. Also, we hope you have fun playing with your solution, which, despite its simplicity, will have an uncanny ability to detect similar web pages. (cos-play?)

Technically speaking, cosine similarity is a measure of similarity between two nonzero vectors of an inner product space that is the cosine of the angle between the vectors, but don't let that definition scare you. It just measures the similarity of two arbitrary objects, and it works for any objects that can be represented as a bunch of attribute-value pairs. It's often used to compare text documents, with the attribute-value pairs being the word frequencies. For example, suppose one document is "if it is to be it is up to me to do it". The corresponding frequency vector is {be=1, do=1, if=1, is=2, it=3, me=1, to=3, up=1}.

A great thing about cosine similarity is that it's quick and easy to calculate, and can detect similarities for many different kinds of objects, from text to Pokémon to used cars. It's commonly used in data mining and machine learning.

The cosine similarity of two vectors A and B is:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \, \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}},$$

where A_i and B_i are components of vectors A and B respectively.

Again, don't let this definition scare you. To make this concrete, consider the cosine similarity of "if it is to be it is up to me to do it" and "let it be". The attribute vector of "let it be" is {be=1, it=1, let=1}. Because the two strings only have "be" and "it" in common, the numerator is just $1 \cdot 1 + 3 \cdot 1$, the sum of products of the common word frequencies. The denominator is $\sqrt{1^2 + 1^2 + 1^2 + 2^2 + 3^2 + 1^2 + 3^2 + 1^2} \sqrt{1^2 + 1^2 + 1^2} = \sqrt{27}\sqrt{3} = 9$, and the cosine similarity is therefore 4/9, or approximately 0.44.

Note that the numerator is only influenced by the words that appear in *both* documents. Because the numerator is the sum of products of the frequency of each word in the documents, a word that is missing from one document (frequency 0) does not affect the numerator's sum. The value of the similarity varies from -1 to 1, or 0 to 1 if the attribute values are non-negative (as they are for document word frequencies). The cosine similarity of a document and itself is 1, and the cosine similarity is 0 for two documents that contain no words in common.

Part 1: A `Document` class

Write a `Document` class with:

- A public constructor that takes a URL string such as `"https://en.wikipedia.org/wiki/Shiba_Inu"`.
- An instance method that takes a second `Document` and returns the cosine similarity, calculated using the formula above.
- A `toString` method that overrides `Object.toString` and returns a short string that identifies the URL represented by the document.

Some hints:

- Use the `Scanner` class to process a web page a word at a time, to build a frequency table for words in the document. You can construct a `Scanner` from a URL string:

  ```
  Scanner sc = new Scanner(new URL(urlString).openStream());
  ```

  Use the `Scanner`'s default tokenization, and don't bother cleaning up the input. This will keep your program simple, and cosine similarity is tolerant of noisy data. For this assignment, do not catch the exceptions thrown by the `URL`; just declare your constructor or methods to propagate them outward.
- Use double-precision floating point arithmetic to compute the numerator and denominator for cosine similarity even though they are integers; integer arithmetic could overflow, yielding wildly incorrect results.
- As an optional optimization, a `Document` can cache the sum of the squares of the frequencies, which will speed up the computation of cosine similarities.

Part 2: The closest match in a set of documents

Write a program called `ClosestMatch` that takes an arbitrary number of URLs on the command line and prints the two URLs for the most similar pair of web pages. Do not bother handling any exceptions thrown by the `Document` class; just declare your `main` method to propagate them outward. For n documents your program should perform $n(n-1)/2$ calls to the cosine similarity method. In other words, each web page should be compared to every other web page exactly once. Cosine similarity is (theoretically) symmetric, so there is no need to calculate *docX*'s similarity to *docY* if you've already computed *docY*'s similarity

to *docX*. We say "theoretically" because, on a real computer, there is imprecision in floating point arithmetic.

Test your program by running it with approximately six Wikipedia articles consisting of two articles on closely related topics and four articles on unrelated topics (e.g., two animals, a plant, a corporation, a rock band and a painter). See if the program can find the two related articles.

Part 3: The closest match to each document in a set

Write a program called `ClosestMatches` that takes an arbitrary number of URLs on the command line and finds the closest matching web page for *each* of the command line arguments. With n command line arguments the program should print `n` pairs of URLs: one for each URL and it's closest match, with each pair on its own line. This program should perform the same $n(n-1)/2$ calls to the cosine similarity method as in Part 2, but save the results in an appropriate data structure so that the closest match for each URL can be determined. As in Part 2, don't bother handling exceptions.

Test your program by running it with approximately ten Wikipedia articles drawn from five subject areas (two articles per subject area). See how many of the pairs your program correctly matches.

Evaluation

Overall this homework is worth 50 points. To earn full credit your solution must demonstrate correct basic use of some Java collections, must build on Travis CI using our Gradle and Checkstyle build configuration, must use proper access modifiers (`public`, `private`, etc.) for your fields and methods, must include descriptive Javadoc comments for all public methods, and must follow the Java code conventions. Hint: use **Ctrl** + **Shift** + **F** to auto-format your code.

We will grade your work approximately as follows:

- Successful use of Git, GitHub, and build automation tools: 5 points
- Basic proficiency in Java: 20 points
- Fulfilling the technical requirements of the program specification: 15 points
- Documentation and code style: 10 points