Homework #1: Cosines o' the Times

Due Thursday, January 26 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.

Preliminaries

During this course you will become familiar with several popular (and industry-relevant) software engineering tools including Eclipse, Git, Gradle, Travis CI, FindBugs, and Checkstyle, in addition to Java. Before you start working on the assignment, make sure your environment is set up correctly. If you have problems with the course infrastructure, reach out to the course staff as soon as possible through Piazza or office hours.

Git and GitHub

Git is a distributed version control system commonly used for large software projects, and GitHub is a hosting service for Git repositories. We will be using GitHub and Git to distribute homework assignments, for you to turn in your homework, and for us to give you grades and other feedback on your work. The basic idea is that you will clone – or make a copy of – your GitHub repository on your local computer. You can then pull changes from GitHub to receive our feedback and any new homework assignments, work locally on your own computer to complete your homework, and commit and push your completed homework back to GitHub, so we can grade it.

Setting up your repository

To set up your repository, sign up for a GitHub account; you may use your existing GitHub account, if you have one. Then fill out the web form here:

`http://garrod.isri.cmu.edu/214/registration`

After filling out that form, confirm your email. Upon confirmation, a GitHub repository will be set up for you and you will receive an email from GitHub asking you to join the 15214 organization. If you do not get this email, go to https://github.com/CMU-15-214/ and join the organization.

Make an initial clone of your Git repository with:

```
$ git clone https://github.com/CMU-15-214/your-andrew-id.git
```

where *your-andrew-id* is your Andrew ID. This will create a directory on your local computer, with your Andrew ID as the name. This directory is the copy of your Git repository in which you will work. Note that graphical frontends for Git are available, such as GitHub Desktop, but we recommend to use the command line tools.

Retrieving assignments and grades

If you have already cloned your Git repository to your local computer, you can pull changes from GitHub (to receive new assignments or grades, or work you've committed from another clone of your repository) with:

```
$ git pull
```

Build and Test Automation

Gradle is a build tool that automates building, deploying, and testing projects and takes the best features from both Ant and Maven. Gradle allows you to build, analyze, and execute your code from the command line. It will later be useful to automatically manage dependencies as well. Travis CI is a web service that automatically builds your code and runs tests when you push to GitHub, sending you an email if the tests fail (if you have set up your email for git commits). These tools enable you to more-easily maintain the integrity of your code, ensuring that new code does not break your project.

Checkstyle is a development tool to point out when Java code does not adhere to common conventions. It automates the process of checking for common stylistic mistakes such as the use of magic numbers. For this course, we will be following a subset of the Sun Code Conventions which can be viewed here. Checkstyle is set up to run automatically with Gradle and will cause your build to fail when the guidelines are not followed. You can run Gradle locally or wait for a message from Travis-CI. There are also plugins for IDEs as Eclipse and IntelliJ IDEA if you prefer immediate feedback in the IDE. Checkstyle is not an autograding tool; it is only meant to help you to avoid certain common mistakes.

Development Environments

Integrated Development Environments (IDE) tend to provide a much better experience than classic text editors for writing Java code. There are several IDEs available, but Eclipse and IntelliJ IDEA tend to be the most common ones. In addition to the Gradle build files, we will distribute setups for Eclipse projects (in Eclipse use "Import – Existing Projects into Workspace"), but you can easily import those with other IDEs as well. It is up to you which IDE you use, if any.

Turning in your work

After you are done working within a clone of your repository, you can turn in your work with

```
$ git add file1 file2 dir1 ...
$ git commit -m "Completed homework 1."
$ git push
```

where *file1*, *file2*, *dir1* and so on are the names of files and directories you have added or changed and the commit message (after the `-m`) is an arbitrary message to describe your work.

1. The command (`git add`) instructs git to track changes to a set of files in your clone; this is called adding the files to your *staging area*. If you pass a directory name to `git add` then all the files added or changed in that directory and all subdirectories (recursively) will be tracked and staged.[1] You can check what files are staged with the `git status` command.
2. The command (`git commit`) records all the locally-tracked changes as a new version of the repository, along with a message that describes the new version. You can check all of the recent commits on your machine with the `git log` command.
3. The command (`git push`) records the most-recent committed version to the remote server, your repository on GitHub. GitHub will then automatically trigger a build on Travis-CI.

Your homework is not turned in unless you have completed all three steps. Check on GitHub whether all your files are correctly pushed.

Each new version is essentially a *local* checkpoint of your work, which you can turn in when you next push your repository to GitHub. If you push your repository to GitHub but have not staged and committed your changes, those changes will not be pushed to GitHub.

Whenever you have finished a feature of your homework, though, it is a good practice to commit it to your repository by doing the above. You should commit often and use helpful commit messages. It is common to commit multiple versions locally before pushing your work, although you might want to periodically push your work to GitHub even if your homework is not complete because this essentially makes a backup copy of your work.

[1] Remember that `.` denotes the current directory. So, (`git add .`) will add all files in the current directory to the staging area.

Also, if you attempt to push your repository to GitHub but the GitHub repository has changed since you last ran `git pull`, your push will fail. To fix this you just need to pull the other changes from GitHub (using `git pull`) and attempt your push again.

When you are done pushing your work to GitHub, you should always check GitHub to confirm your expected files are there. Alternatively, you can create a new clone of your repository (using `git clone`) in a new location on your computer, and test your solution in that new location. This method allows you to test exactly what the TAs will test when they clone your repository from GitHub.

Don't push changes to a homework subfolder after the deadline (unless you intend to spend part of your "late" budget). We will use the GitHub timestamp of the latest `git push` event to determine if you were on time.

Additional resources

Here are some resources if you are interested in learning more about git. We recommend anyone who is going into any industrial software development company to at least try the 15-minute tutorial on git. Version control is a very powerful tool in software development, and mastering it will remove many headaches that you may encounter otherwise.

- 15-Minute Tutorial: https://try.github.io/
- Pro-Git Book (free): http://git-scm.com/book/en/v2/
- Github Game (free): http://pcottle.github.io/learnGitBranching/

The Actual Assignment: Cosine Similarity

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. 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". For the first text, the frequency of terms can be represented through the mapping {be=1, do=1, if=1, is=2, it=3, me=1, to=3, up=1}, for the second one as {be=1, it=1, let=1}. If we assume a fixed order of all known words, let's say ('be', 'do', 'if', 'is', 'it', 'me', 'to', 'up', 'let'), we can represent the frequencies for both documents as vectors $(1, 1, 1, 2, 3, 1, 3, 1, 0)$ and $(1, 0, 0, 0, 1, 0, 0, 0, 1)$, to which we can apply the formula. For the numerator, we can ignore all terms that occur zero times in either document, resulting in $1 \cdot 1 + 3 \cdot 1$ for 'be' and 'it'. The denominator is $\sqrt{1^2 + 1^2 + 1^2 + 2^2 + 3^2 + 1^2 + 3^2 + 1^2} \sqrt{1^2 + 1^2 + 1^2} = \sqrt{27}\sqrt{3} = 9$. Hence, the cosine similarity is $4/9$, 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 0 to 1 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.

The homework assignment is divided into three parts, detailed below.

Part 1: A `Document` class

Write a `Document` class (see hints below) 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.

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. 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 should save the results in an appropriate data structure of Java's collection framework 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.

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.

- You do not need to handle exceptions thrown by the `URL`; just declare your constructor or methods to propagate them outward to the main method.

- 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.

- In Eclipse, use **Ctrl** + **Shift** + **F** to auto-format your code; other IDEs have similar functions.

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.

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