Lab #2: Text Similarity

Due Saturday, June 8 at 11:59 p.m.

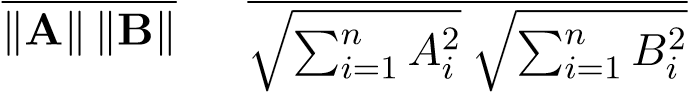
In this assignment you will use [cosine similarity](https://en.wikipedia.org/wiki/Cosine_similarity) to detect similarities between code files. The goals of this assignment are to familiarize you with our course infrastructure, let you practice object-oriented programming with Java, and introduce you to the [collections framework.](https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html) Also, we hope you have fun playing with your solution, which, despite its simplicity, will have an uncanny ability to detect similar code files.

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 code files, with the attribute-value pairs being the word frequencies. For example, suppose one code file 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:

**A**

similarity = cos(*θ*) = = *,*

where *Ai* and *Bi* 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 · 1 + 3 · 1, the sum of products of the common word frequencies. The denominator is , 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* codes. Because the numerator is the sum of products of the frequency of each word in the codes, a word that is missing from one code file (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 code file word frequencies). The cosine similarity of a code file and itself is 1, and the cosine similarity is 0 for two code file that contain no words in common.

# **Problem 1: A CodeFile class**

Write a CodeFile class with:

* A public constructor that takes a URL string such as "20171001234/lab1/turtle/ TurtleSoup.java.java".
* An instance method that takes a second code files 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 CodeFile.

Some hints:

* Use the [Scanner](https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html) class to process a code file a word at a time, to build a frequency table for words in the CodeFile. You can construct a Scanner from a URL string:

Scanner sc = new Scanner(new URL(urlString));

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 CodeFile can cache the sum of the squares of the frequencies, which will speed up the computation of cosine similarities.
* 提示：示例文件sample1.code和sample2.code的相似度为0.939040909148062。

# **Problem 2:** The closest match in a set of code files

Write a program called ClosestCodeMatch that takes an arbitrary number of URLs on the command line and prints the two URLs for the most similar pair of code files. Do not bother handling any exceptions thrown by the CodeFile class; just declare your main method to propagate them outward. For *n* CodeFiles your program should perform *n*(*n* − 1)*/*2 calls to the cosine similarity method. In other words, each code file should be compared to every other code file 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 code files consisting of two files on closely related project and four code files on unrelated projects (e.g., lab1 and lab2 projects). See if the program can find the two related source code files.

# **Problem 3:** The closest match to each CodeFile in a set

Write a program called ClosestCodeMatches that takes an arbitrary number of URLs on the command line and finds the closest matching code file 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 problem 2, but save the results in an appropriate data structure so that the closest match for each URL can be determined. As in problem 2, don’t bother handling exceptions.

Test your program by running it with approximately ten source codes drawn from five different projects (two source code files per project). See how many of the pairs your program correctly matches.

# **Problem 4: LabClosestMatches**

**假设在 codes/lab1/目录下存在以下结构的文件组织：**

├─Java课内实习-201710001234-xxx-实习1

│ ├─Java课内实习-20171000123-xxx-实习1

│ │ └─lab1\_code

│ │ ├─rules

│ │ └─turtle

│ └─lab1\_code

│ ├─rules

│ └─turtle

├─Java课内实习-20171001235-xxx-实习一

│ └─lab1

│ └─lab1\_code

│ └─lab1\_code

│ ├─bin

│ │ ├─rules

│ │ └─turtle

│ ├─rules

│ └─turtle

├─Java课内实习-20171001236-xxxx-实习一

│ ├─rules

│ └─turtle

└─Java课内实习20171001237-xxxx-实习一

└─Java课内实习20171001237-xxx-实习一

└─Java课内实习20171001237-xxxx-实习一

└─lab1\_code

├─123

├─rules

│ └─bin

└─turtle

└─bin

Write a program called **LabClosestMatches,**  实现如下方法：

/\*\*

\* 用于评价各相关目录下，指定文件的相似性。

\* Similarity 子目录1 子目录2

\* 100% Java课内实习-201710001234-xxx-实习1 Java课内实习-201710001235-xxx-实习1

\* 89% Java课内实习-201710001234-xxx-实习1 Java课内实习-201710001236-xxx-实习1

\* ....

\* **@param** path 作业文件所在的目录，比如这里是：codes/lab1

\* **@param** fileNameMatches：用来过滤进行比较的文件名的文件名或者正则表达式.

\* 如 "DrawableTurtle.java"，"\*.java","turtle/\*.java"

\* 如果一个子目录下有多个符合条件的文件，将多个文件合并成一个文件。

\*

\* **@param** topRate:取值范围从[0,100],输出控制的阈值

\* 从高往低输出高于topRate%相似列表，如

\* **@param** removeComments:是否移除注释内容

\* \*/

**public** **static** **void** closestCodes(String path, String fileNameMatches,**double** topRate,**boolean** removeComments)

**注意：**

1. **代码需要有良好的组织。**
2. **需要有足够的单元测试代码。**
3. **Problem2-3中的2个类，需要先定义接口，再写实现。以支持后期代码扩展。**