Natural Language Detection

Welcome to the final project for CSC211. **Read this document thoroughly before starting on your project.**

Your task is to implement a machine learning algorithm that detects what language a given text file is written in. You will do this by employing a method called **n-gram frequency analysis**. We will restrict our analysis to languages that use the english alphabet.

Machine Learning

When we use statistics in computer science, this is called **machine learning**. It is a beautiful field yet in its relative infancy. Using the computational power at our fingertips, we use machine learning (a.k.a. statistical models) to perform a variety of tasks. These tasks range from predicting disease spread in a population to finding out which ads to present to you to best catch your attention. Voice recognition software like Siri and Cortana use machine learning to better understand what people are saying. We can use machine learning to look and pictures of someone's face and guess their age with remarkable accuracy. The variety of tasks that can be tackled with this approach is limited only by your imagination.

Any machine learning project is implemented with a 2-step process. The first step is to **train** an algorithm on a provided **sample** of data. In our case, you will be provided with some text files in various languages. Your algorithm will train on those files. The second step is to test your **trained algorithm** on some files that contain text different from that in the training files. Done correctly, your algorithm will predict, with remarkable accuracy, the language that the test file is written in.

n-gram Frequency Analysis

An **n-gram** is a sequence of n consecutive characters from a given sample of text. For this project, you will take $\mathbf{n = 3}$. From here on out, we will call a sequence of 3 consecutive characters a **trigram**.

The idea behind **trigram frequency analysis** is that you measure the **frequency** of each trigram in a given text file. You will count how many times each trigram occurs in a given text file and you will store these frequencies in a vector. We will call such vectors **frequency vectors**.

You will calculate a frequency vector for each of your training languages. You will then compute the frequency vector for the test language. You will then calculate the **similarity** between the test frequency vector and each of the

training frequency vectors. The training frequency vector that yields the highest similarity will correspond to the best matching language.

Project Structure and Deadlines

This project will be split into two milestones.

Milestone 1

Due: Thursday April 19^{th}, 2018 at 11pm

For Milestone 1, you will write an algorithm to compute the frequency vector from a given text file.

Trigrams

As stated earlier, a **trigram** is a sequence of three consecutive characters from a given string. For example, given the string "hello", the trigrams are "hel", "ell", and "llo". When calculating what the trigrams are, you will consider upper and lower case letters to be the same. You will also ignore any characters that are **not letters**. For example, given the string "This is I.", the trigrams are "thi", "his", "isi", "sis", and "isi".

Trigram Frequency Vectors

In order to compute the similarity between two documents, it is necessary to represent the trigram frequencies as vectors in the mathematical sense. Specifically, these vectors need to have the **same number of elements** (i.e. they need to have the same dimension) and **those elements need to be in the same order** across different vectors. While we want you to have great freedom of design in this project, we must also strive for correctness. As such, we must standardize the ordering for trigrams.

In class, we discussed an algorithm for representing any trigram of ASCII characters as an element in a vector 256^3 elements long, using base-256 encoding. However, since we are only concerned with characters in the english alphabet, we can get away with base-26 encoding. (Remember that we are ignoring any characters that are not letters, and are treating upper-case letters as if they were lower-case letters.) The unique charracters under consideration are **a-z**. Therefore, we can use vectors that are only 26^3 long. Thus, the encoding you must use is that a maps to 0, b maps to 1, c maps to 2, ... , z maps to 26. With this encoding, the ,trigram "aaa" would map to 0 and the trigram "zzz" would map to 17,575 (which is 26^3 - 1). As a more useful example, consider the string "This is I.":

- The trigram "thi" maps to the index $19 * 26^2 + 7 * 26 + 8 = 13,034$. Its frequency is 1.
- The trigram "his" maps to the index $7 * 26^2 + 8 * 26 + 18 = 4,958$. Its frequency is 1.
- The trigram "isi" maps to the index $8 * 26^2 + 18 * 26 + 8 = 5,884$. Its frequency is 2.
- The trigram "sis" maps to the index $18 * 26^2 + 8 * 26 + 18 = 12,394$. Its frequency is 1.
- All other indeces in the frequency vector would contain 0.

Important note: the leftmost letter in the most significant digit in our base-26 system.

Milestone requirements

Your program will receive a single command line argument that is the name of a text file. Your program should start by reading this text file. It will then calculate what the trigrams are and will store the frequency of eacy trigram in some way. How you go about storing these frequencies is entirely up to you. You could use C-arrays, or the C++ **std::vector** class, or your own class, or anything else that makes sense.

You must name your output file **frequencies**.

Sample command-line input:

```
$ ./frequencies german
```

Your program should print out the integer values of the frequencies in the order discussed in the section above. The integer values need to be **space-separated** and there needs to be a **new-line character** after the frequency of the very last trigram.

Grading Rubric

- Functional Correctness: 70%
- Design, Representation, and Comments: 30%

Milestone 2

Due: Monday April 30^{th}, 2018 at 11pm

You will calculte a trigram frequency vector for each file in a given set of training files. You will also calculate the frequency vector for a test file. You will then compute the similarity between the test frequency vector and each training

frequency vector. You will keep track of which similarity was the greatest and will output the name of training file which produced the best match.

Similarity

Given two vectors (**mathematical vectors, not C++ vectors**), the similarity between those vectors can be described using **cosine similarity**. This is the cosine of the angle between the two vectors.

A formula for the **cosine similarity** of two vectors A and B, where both A and B have n elements, is:

$$\cos^2\theta = \frac{\left(\sum_{i=0}^{n-1} A_i B_i\right)^2}{\left(\sum_{i=0}^{n-1} A_i^2\right)\left(\sum_{i=0}^{n-1} B_i^2\right)}$$

This may look like a scary equation but, when we get down to it, it is actually pretty simple. Let's deconstruct it and look at its parts.

Let's take a look at the numerator:

$$\sum_{i=0}^{n-1} A_i B_i$$

This is the **sum** of the **element-wise product** of **corresponding elements** in the vectors A and B. In math, this is known as the **dot-product** of two vectors.

Let's take a look at each term in the denominator:

$$\sum_{i=0}^{n-1} A_i^2$$

This is the **square-root** of the **sum** of the **element-wise square** of **each element** in the vector A. In math, this is called the **norm** of a vector. The same goes for the second term in the denominator.

The value for cosine similarity will range between 0 and 1 inclusive. The larger this value is between two vectors, the more similar they are.

Pitfalls

As it so often happens, there are some pitfalls when calculating the similarity between vectors using built-in C++ integer types. The frequency vectors will

have some values that are quite large and when these values are squared, they **overlow** even unsigned long long. Naturally, we are going to have you use **bigints** to get around this issue. You will store the intermediate parts of the similarity calculation as bigints and then use bigint methods to calculate $cos^2\theta$.

Another thing to notice is that similarity is going to be of type `double`. We have not implemented bigints that can handle floating-point types. We will get around this problem by using a little math trick called **scaled-division**. The idea is that you:

- take the numerator and multiply it by some convenient number (e.g. $1,000,000$),
- perform bigint division using the **scaled numerator**,
- convert the result into a regular integer,
- and finally perform **floating-point division** by 1,000,000

to get the actual value for $cos^2\theta$. You can then calculate the square-root of this value to get the **cosine similarity** between the two vectors.

Note that you will be performing multiplications and divisions with some really large numbers. The iterative bigint `multiply` and `divide` methods you implemented are going to be painfully slow for numbers of this magnitude. We suggest that you download the solution key for bigint provided to you and use the `fast_multiply` and `fast_divide` methods implemented by a certain wonderful TA.

Milestone Requirements:

Your program will receive an unknown (greater than 2) number of command-line arguments. These will be the names of the training files and finally the name of the test file. You are to calculate the **trigram frequency vector** for each of the files given. You will then compute the **cosine similarity** between the frequency vector of the test file and the frequency vector of each training file. You will find the highest value for similarity and then print the name of the language that was the best match (followed by a new-line character).

You must name your output file **language**.

Sample command-line input:

```
$ ./language english german spanish icelandic maori test
```

If `icelandic` happened to be the best match for `test`, your command-line should end up looking like:

```
$ ./language english german spanish icelandic maori test
icelandic
$
```

Grading Rubric

- Functional Correctness: 70%
- Design, Representation, and Comments: 30%

Important Note: You are not being graded on how fast your program runs, with one caveat: if your program is so slow or so memory-inefficient that Mimir cannot run it, you will receive no points for functional correctness. A well-written program should run in anywhere from 5-60 seconds on the data set we have provided. If your program takes multiple minutes to run on your conputer, then something is wrong.

Getting Started

We provide (via GitHub) a large data-set in addition to this document and some started code. You can obtain it via:

```
$ git clone https://github.com/csc211/final-project
```

You will notice that inside the resulting directory are two sub-directories called `training_languages` and `testing_languages`. Each sub-directory contains some text files, each named for a language. The text in the training files is different from the text in the testing files.

Helpful Hints

Dealing with the data files

If you are anything like some of your instructors, you hate typing repetitive and unnecessary things on the command line. You should learn to rely on *wildcards*, also known as *file globbing*. For example, if you ran:

```
$ ./language training_languages/* testing_languages/english
```

the command line will expand the * into *all the files in the* ***training_languages*** *sub-directory*, so your program will see every file individually just as if you had painstakingly typed out the name of every file individually!

Writing your own compile script

For milestones 1 and 2, you need to write your own compile script. We have provided you something to start you off but you will need to edit it. In particular, you need your compile script to take **every .cpp file** you have written and compile them into a **single executable file**.

Note that ease-of-debugging and speed-of-execution are at odds with one another. When you are just developing, you **should not** have flags like `-O2` (which stands for optimization level 2), and you **should** have a flag like `-g` (which keeps some symbol information so your debugger can show you your own source code). However, prior to submitting your project, you should remove the `-g` flag. You can also experiment with putting in the `-O2` od `-O3` flags for compiler optimization. You may even want to see which optimization flags produce the fastest executables. Sometimes, the difference can be significant!

You can put a stopwatch to your program with the `time` command:

```
$ time ./language training_languages/* testing_languages/english
```

On my (rather fast) laptop, I can say something like this:

```
$ time ./language training_languages/* testing_languages/english
training_languages/english

real 0m2.610s
user 0m2.586s
sys 0m0.024s
```

The relevant number here is the `user` time, since the computer is also busy doing some other stuff (like running my operating system). Essentially, my program ran in 2.586 seconds.

Memory Efficiency

If you are not careful, it is easy to make your program use too much memory or run very slowly (the two are often related). Remember that arguments are passed to functions by **value** in C++ by default. This means that if you take a vector containing ~20,000 8-byte integers and pass it around by value, C++ is making several copies of that vector. Each copy is worth ~160 kB. These can add up very quickly, particularly given that the total size of the language text files is ~200 MB. Use the information presented in class on passing arguments by reference to cut down on this wasted memory usage.

Good Luck!