CSCI 103L Spring 2019 Programming Midterm: “City Similarity”

Thursday, March 28th 2019 7:00pm - 9:00pm

Problem Introduction

Data science and machine learning are very hot topics in computer science. Much of what practitioners are able to do seems like magic, when really they are often the result of applying simple algorithms to the right data. This programming midterm will expose you to one such algorithm: the cosine similarity.

One broad class of problem data scientists like to solve involves similarity, i.e comparing two things and returning some objective notion of how “similar” they are. For examples think of dating sites (how “similar” and thus compatible are two people), shopping (if you like item A, what are other similar items you might buy?), and movies/entertainment (Netflix recommendations).

At first, doing such comparisons seems difficult, especially in an objective, mathematically sound manner because there are many facets (or features) to such a comparison. The trick is to create what is called a **feature vector**, which is nothing more than a set of numbers, one per feature. Each number is chosen or designed in some way to capture the salient information of a particular feature. For a dating site, height is easily turned into a feature. For movies the average rating might be a good feature.

Thus, the dataset for making comparisons comprises the feature vector for each item (person, product, movie, etc.) in our data.

To make a comparison there are many techniques, however for this assignment we have chosen a similarity metric known as the cosine similarity. Our dataset will be a list of cities and a **very simple** feature vector. Your task will be to read in the data, implement cosine similarity and produce recommendations on which cities are similar to each other.

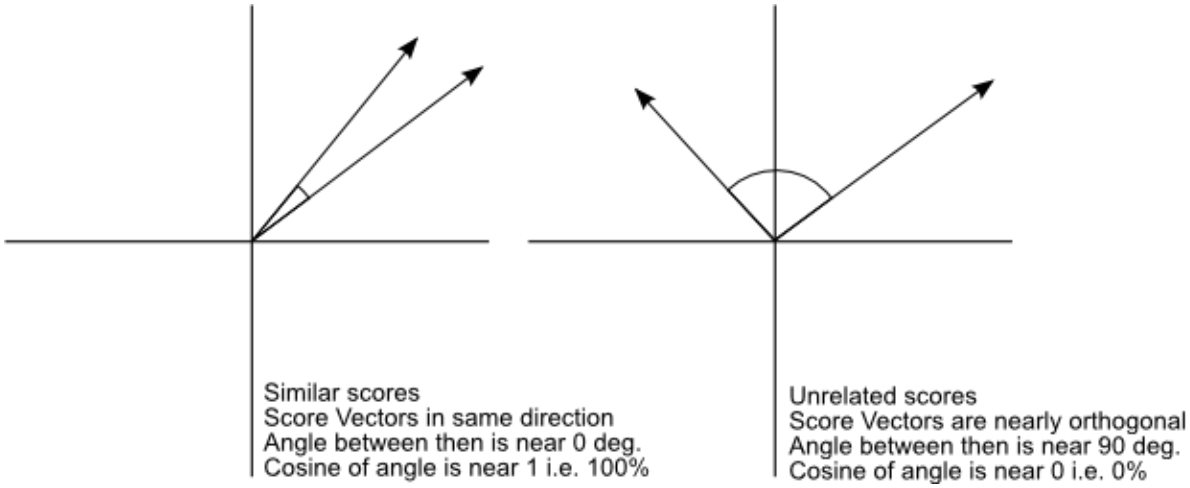
Cosine Similarity

We call the data that represents each item a feature vector because it is exactly that, a vector in the multidimensional feature space. For example here is the feature vector for two cities:

New York, NY: <1, 0.2169543439, 1, 0.5331386861>

Los Angeles, CA: <0.4638639785, 0.2199853237, 1, 0.4616058394>

Once we have the notion of a vector, we can consider the angle between two vectors. We are using the cosine of this angle as the cosine similarity.



In the figures above we show the vectors (in a 2D space) that represent two different items, the cosine similarity is found by calculating the cosine of the angle between the two vectors. Using trigonometry we can derive the formula for cosine similarity as:

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

The formula for dot product will be explained further below. For this assignment we will be working with a 4 dimensional space, however in practice the formula is the same regardless of the number of dimensions. The idea is that two items with similar feature vectors will have a small angle between them, resulting in a cosine similarity close to 1.0. Two items with very different feature vectors will have a large angle and a cosine similarity close to 0. Thus to compare the similarity of two cities, we apply the formula using the feature vectors of each city. As an example we can do Los Angeles and New York (numbers shortened for clarity):

LA · NY = 0.4638*1 + 0.2199*0.2169 +1*1 + 0.4616*0.5331 = 1.7577

||LA|| = sqrt(0.4638^2 + 0.2199^2 + 1^2 + 0.4616^2) = 1.2152
||NY|| = sqrt(1^2 + 0.2169^2 + 1^2 + 0.5331^2) = 1.5269

cos(θ) = (LA · NY)/(||LA||*||NY||) = 1.7577/(1.2152*1.5269) = 0.9473

This result being close to 1.0 indicates fairly similar cities.

One further note, by looking at the formula for cosine similarity we can observe that the magnitude of a vector ||A||, can be computed as the square-root of the dot product of the vector with itself: ||A|| = sqrt(A ˙ A). This means we really only need one function (the dot product) to compute the cosine similarity.

City Similarity

For this assignment we will be comparing cities. Perhaps you're looking to move to a different city, and you would like to know which other cities are similar to a few cities you like. We can do this if we come up with a feature vector that captures the essence of a city. The feature vector for this assignment will be **very** simple so we can understand the comparisons that are being made. It is a bit of a toy example, but still makes reasonable comparisons.

The feature vector is composed of the following numbers for each city:

Normalized Population (i.e relative population, largest city = 1.0)
Normalized Median House Price (relative to most expensive = 1.0)
Coastal State (boolean Yes = 1, No = 0. Coastal: e.g CA, WA, TX, FL, NY, etc)
Normalized Trump Vote (relative to highest vote = 1.0)

Datafile

The data file containing city names and feature vectors is very simple:

Line #1: Integer N = number of cities in the file
Lines #2-(N+1): CityName <norm pop><tab><norm house value><tab> <coastal><tab> <trump vote>

The feature vector values are floating point and will be read in as type **double**. Here is an example for a file with 4 cities:

```
4
NewYorkNY    1    0.2169543439    1    0.5331386861
LosAngelesCA 0.4638639785    0.2199853237    1    0.4616058394
ChicagoIL    0.315034807    0.07290304055    0    0.5658394161
HoustonTX    0.268212687    0.05921577386    1    0.7624817518
```

Please note that the city names have been processed to remove spaces and concatenate the state. The cities are unique.

City struct

This assignment makes use of a simple struct object to contain data about each city. It is defined for you as:

```
struct City {
    string name; //name of the city
    double v[4]; //feature vector read from datafile
    double sim; //used to hold similarity for comparison
};
```

Programing Instructions

The overall task for your program is two-fold: 1) read in a dataset with city names and feature vectors. 2) Given a city name, return the 10 best matches in the dataset.

In order to make the programming midterm as straightforward as possible we will not ask you to implement any functionality in main(). All functionality will be implemented through a series of functions. This also means that you will be able to test each function individually.

Once you login to Vocareum, you will see several files. Most important to review are citysim.h and citysim.cpp.

You will do all of your work inside citysim.cpp.

DO NOT MOVE OR CHANGE ANY OF THE #define, #ifdef or #end preprocessor directives. These directives make is possible for the test code to test your functions individually. **NULL** is a special pointer value (it is an alias for the value zero) that means “this pointer points to nothing”.

Read the descriptions below and complete each function. The **SLOC** (source lines of code) number tells you how many lines of code were used in the reference solution. Use this a a guide as you complete your implementation.

Function #1: `double dot4(double v1[], double v2[]) (6 SLOC)`

This function computes the dot product of two length 4 vectors (represented as arrays). Returns this value as a double. The formula for dot product is (given two vectors A, B, with elements Ai, Bi):

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

Function #2: `double len4(double v1[]) (1 SLOC)`

This function computes the magnitude (i.e length) of a 4 element vector (represented as an array). The formula for magnitude of a vector A is:

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

Which can be seen to be the square-root of the dot product of the vector with itself.

Function #3: `double cossim(double v1[], double v2[]) (1 SLOC)`

This function computes the cosine similarity between two length 4 vectors (represented as arrays). The formula for cosine similarity can be expressed in terms of the dot product and magnitude of two vectors A and B as:

$$\frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

You can assume that there exist no vectors of length zero in our dataset.

Function #4: `City* read_cities(const char* fname, int* n) (19 SLOC)`

Arguments:

`const char* fname`	C-string with the name of the data file to open
`int* n`	Pointer to an integer. Use dereferencing to return the number of cities read from the file.

Return value:

`City*`	Pointer to an array of city objects. You must allocate this array dynamically using **new**. If an error is encountered the pointer returned must be **`NULL`**.

This function will read in the city names and feature vector from the given file. You must do the following:

#1: Attempt to open the file. If this fails, return the **`NULL`** pointer.
#2: Assuming the file opens, read in the number of city entries. You must check to see if this step fails. If this fails, return the **`NULL`** pointer.
#3: Assuming the number of entries in the file can be read, use dereferencing to return this value.
#4: Allocate an array of **`City`** objects using new of appropriate size.
#5: Loop over the remaining lines of the data file reading in each city name and feature vector into an item in the array of **`City`** objects. Make sure you initialize the **.sim** data member of each city object to 0.0 at this point.
#6: Return the pointer to the array of **`City`** objects

If the first item read from the file is an integer (the number of cities), then you can assume the rest of the file is formatted properly.

Function #4: `City* find_city(City* cities, string name, int n)` (8 SLOC)

Arguments:

`City* cities`	Pointer to an array of City objects
`string name`	C++ string containing the name of the city to find
`int n`	Length of the array of City objects

This function will search the array of City objects for one that matches the input parameter **name**. If the City is found in the array, return a pointer to that item. If the City is not found in the array, return **NULL.**

Function #5: `void sort_cities(City* cities, int n)` (9 SLOC)

Arguments:

`City* cities`	Pointer to an array of City objects
`int n`	Length of the array of City objects

This function will sort the array of City objects based on the data member **sim**. Implement a reverse sort, i.e largest to smallest (after sorting, cities[0].sim should be the largest similarity value). It is suggested that you implement bubble or selection sort. You may use the `swap( )` function provided in `<algorithm>` as follows (indexes will vary based on **your** code):

```
swap(cities[i], cities[j]);
```

Function #6 `void suggest(City* cities, string name, int n)` (9 SLOC)

Arguments:

`City* cities`	Pointer to an array of City objects
`string name`	C++ string with the name of the city to generate suggestions for
`int n`	Length of the array of City objects

This function will generate the suggestions (i.e. closest matching cities) to the one given in the input parameter **name**. To implement this function you should first find the City object in the array that corresponds to the one given in **name** (you may assume that the city is in the array). Then compute the similarity between the given cities feature vector and every other city in the array (except do not compare the city to itself, set the similarity to itself to 0.0). Store the similarities in the **sim** data member. After computing the similarity for all cities, sort the list of cities.

Exam Tips/Tricks

Please read the following list carefully! It will help you complete the exam as smoothly as possible!

- We have provided a Makefile so you can build and test your code, you must use it!
- Type ‘make’ to compile your code.
- Type ‘make test’ to compile and test your code, and to receive your score.
- Note: ‘make test’ can only be run on Vocareum.
    - You can download the code to your laptop for Xcode or similar, but you won’t be able to run ‘make test’.
    - However, ‘make’ will still work.
- DO NOT DELETE any file in your work directory that comes from the starter code. If you do so, ‘make test’ will break and your code may not autograde properly.
- Grading on submission is turned off for this exam! This means you will not see your grades update during the exam. We will be batch grading the exam later.
- However: ‘make test’ will give you your score. Please use ‘make test’ as often as necessary to see how you are doing on the exam.
- You do not need to complete the functions in any particular order. They are all tested completely independently. For example, this means you can complete **suggest()** even if your **sort_cities()** doesn’t work. Or, you can complete **cossim()** even if your **dot4()** or **len4()** functions don’t work.
- Please ‘Submit’ after you complete each function. DO NOT WAIT UNTIL THE LAST MINUTE TO SUBMIT.
- DO NOT WAIT UNTIL THE LAST MINUTE TO SUBMIT.
- DO NOT WAIT UNTIL THE LAST MINUTE TO SUBMIT.
- And if we weren't clear:
    - **DO NOT WAIT UNTIL THE LAST MINUTE TO SUBMIT**

Example Program Executions

First run ‘make’ to build the program. The executable created for testing is called **main**

Here we see the suggestions for Los Angeles with the ‘`top10.tsv`’ file. Of course we see all ten cities in the output.

```
ccc_v1_w_MDUzN_92107@runweb14:~$ ./main top10.tsv
Found 10 cities in file top10.tsv
Enter city name for suggestions (or type 'exit' to quit):
LosAngelesCA
The top 10 best matches to LosAngelesCA are:
SanDiegoCA 0.969855
HoustonTX 0.95222
NewYorkNY 0.947338
SanAntonioTX 0.936052
DallasTX 0.933215
ChicagoIL 0.534587
PhoenixAZ 0.48142
PhiladelphiaPA 0.475426
LasVegasNV 0.439909
LosAngelesCA 0
Enter city name for suggestions (or type 'exit' to quit):
exit
```

Here we see the suggestions for Las Vegas, NV when using the ‘`top50.tsv`’ file.

```
ccc_v1_w_MDUzN_92107@runweb14:~$ ./main top50.tsv
Found 50 cities in file top50.tsv
Enter city name for suggestions (or type 'exit' to quit):
LasVegasNV
The top 10 best matches to LasVegasNV are:
AlbuquerqueNM 0.999771
ColoradoSpringsCO 0.999547
MesaAZ 0.999364
MinneapolisMN 0.999324
NashvilleTN 0.999078
TucsonAZ 0.998624
ColumbusOH 0.997561
MilwaukeeWI 0.99715
IndianapolisIN 0.997075
OmahaNE 0.996689
Enter city name for suggestions (or type 'exit' to quit):
exit
```

Test Descriptions

Run ‘make test’ to see which tests you are passing.

`Testing cossim()`

Tests to see that the correct value is returned when calculating the cosine similarity of two length 4 vectors.

`Testing dot4()`

Test the see that the correct value is returned from the dot4() function (i.e dot product of two length 4 vectors).

`Testing find_city()`

`.find_city1` Tests to see that a pointer to a known city is returned.

`.find_city2` Tests to see that NULL is returned for an unknown city. i.e find_cities() is called with a name that is not found in the data file.

`Testing len4()`

Tests that the proper value is returned from `len4()` (i.e the magnitude of a length 4 vector).

`Testing read_cities()`

`.read_city1` `read_cities()` called with bad file name. Must return NULL.

`.read_city2` `read_cities()` called with empty file. Must return NULL.

`.read_city3` `read_cities()` called with ‘top700.tsv’. Must return correct cities array.

`Testing sort_cities()`

Tests to see if sort_cities() properly sorts the cities array in descending order based on the similarity value.

`Testing suggest()`

`.suggest1` `suggest()` called with “WestNewYorkNJ”. Tests to see if top 10 suggested cities are correct.

`.suggest2` `suggest()` called with “LosAngelesCA” followed by “SanDiegoCA”. Tests to see if .sim data is properly reset or initialized for each search.