CSE 232 Spring 2019

Programming Project #7

In this assignment we will do various manipulations of maps, sets and vectors.. The assignment is worth 50 points (5% of your course grade) and must be turned in before 11:59 on Monday March 18th.

Description

Pattern recognition, related in many ways to machine learning, is of course a big deal these days. We are going to look at a relatively simple and straight forward approach to pattern recognition, document similarity. If I give you two documents, can you tell how "similar" they are to each other.

We will look at two measures: Jaccard Similarity and Cosine Similarity using term frequencies.

Jaccard Similarity

Jaccard similarity is a measure that uses only the unique words found in the documents. The count of the number of times a word occurs in a document is not used. Based on the unique words found in each of two documents A and B, we measure the Jaccard Similarity as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Where the $|A|$ indicates the count of the number of unique elements in A. This similarity results in a real number between 0 and 1, where 1 indicates the documents are identical and 0 indicates the documents are completely dissimilar. For example given the two short documents

A) "This is a test."

B) "This test bothers me. This test bothers you."

We process the words to remove non alphabetic chars and lowercase everything to create:

`|A| == 4` (this,is,a,test), `|B| = 5` (this,test,bothers,me,you), `|A∩B| == 2` (this,test)

`2/(4 + 5 - 2) == 2/7 ~= 0.2857`

Cosine Similarity

Cosine similarity depends not only on the words themselves but their frequency as well. The frequencies that are used in the calculation can vary, but a common one is called the normalized term frequency. If we used just straight frequencies, then documents with many repeated words would tend to dominate the process. To counter this we measure the term frequency by normalizing their values. We do so by first calculating the norm factor for the frequencies of each term in the document

$$NF = \sqrt{(term_a^2 + term_b^2 + \cdots term_n^2)}$$

which is the square root of the frequencies of each word squared. The normalized term frequency is then the $^{term}/_{NF}$ for each term in that document

Using the same two sentences from above now with frequencies, we calculate the normalized frequency for each term.

term	this	is	a	test	bothers	me	you	NF
A	1	1	1	1	0	0	0	$\sqrt{4}$=2.0
B	2	0	0	2	2	1	1	$\sqrt{14} \cong 3.74$
A normalized	½=0.5	0.5	0.5	0.5	0	0	0	
B normalized	2/3.74=0.535	0	0	0.535	0.535	1/3.74=0.267	0.267	

The cosine similarity formula is

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

What's nice about that is that because the A and B values are normalized, the denominator square roots are both 1, meaning they fall away. What is left is to calculate the product of the normalized value pairs and add them all up. Skipping any term with a 0 (which under multiplication makes it 0) we use the normalized pairs for "this" and "test"

$$(0.5 * 0.535) + (0.5 * 0.535) = 0.535$$

a different value then found with Jaccard

Program Specifications

The functions are all listed in the provided `proj07_functions.h` file, but here they are again:

```
ostream& print_vector (ostream &out, const vector<string> &v)
```

print the contents of the vector <string> to the parameter ostream (look at Mimir test cases for format). Return the ostream used.

```
string clean_word(const string&)
```

Take in a string. Create a new string where you remove any non-alphabetic characters and lower case everything. Return that new string.

```
vector<string> split(const string &s, char delim=' ')
```

You already did this in lab. Take in the string, return a vector where the string is split by the parameter character `delim`. Note the default is only provided in the header, not the definitions.

`bool process_words(map_type&, string);`

Takes in an empty map and a string representing a file name. The function opens the file, reads the file one line at a time, splits the line, cleans each word and then records it in the map where the key of the map is the string and the value of the map is how many times that string occurred.

Error: if the file represented by the string cannot be opened, the function returns false and makes no changes to the map. Otherwise the map is updated as indicated and the function returns true.

`ostream& print_map(ostream& out, const map_type& m)`

Like print_vector above, print the contents of the map to the parameter ostream (look at Mimir test cases for format). Return the ostream used.

`double jaccard_similarity(const map_type &m1, const map_type &m2)`

Takes in two maps (representing the result of `process_words` on two documents). Using *only the keys* it calculates the jaccard similarity as described. Returns the similarity.

`double calc_norm_factor(const map_type &m1)`

For a map created by `process_words`, calculate the norm factor of the map/document as shown above (the square root of the frequencies squared). Return the norm factor.

`double cosine_similarity_tf(const map_type &m1, const map_type &m2)`

Calculate the term frequency cosine similarity for two maps created by `process_words` as described above. Return the similarity

Deliverables

You will turn in one file: `proj07_functions.cpp`. We provide you only with `proj07_functions.h` , you must write your own main to test your functions. Mimir can test the individual functions without a main program but it's a good idea for **you** to test your own code with a main, perhaps in the manner that we did previously.

Remember to include your section, the date, project number and comments and you ***do not provide*** main.cpp. If you turn in a main with your code Mimir will not be able to grade you.

1. Please be sure to use the specified file names
2. Always a good idea to save a copy of your file in your H: drive on EGR.
3. Submit to Mimir as always. There will be a mix of visible and not-visible cases.

Notes

1. You turn in the functions only. To test against your own main you can write a separate file

with the main and then compile the two files at the same time. See the lab and videos for examples.

2. For jaccard, you only need to work with the keys. You can calculate set intersection yourself, but much easier to use the algorithm set_intersection, perhaps on two vectors that have the keys of the two dictionaries. Beware that set_intersection requires sorted elements (if you extract the keys into a vector, would they already be sorted or not?). Furthermore, beware of integer division here. Perhaps static_cast<double> is your friend?
3. For cosine, you need to make a new map/vector where the element frequencies are normalized. Then wherever both documents have the same key, you need to multiply the two normalized frequencies and add those products up. Here set_intersection probably does not do what you want.