Computer Science I — Calculating Text Statistics

CSCI-141 Project

version 3 11/12/2023

1 Problem Description

Google Books is a service offered by Google that allows a user to search a digital database of books and magazines that Google has scanned and converted to text using optical character recognition (see `https://en.wikipedia.org/wiki/Google_Books` for more details). Data exist for years as early as the 1500's. The most recent data available are from the year 2008.

One of the tools that Google has developed to mine this data is the Google Ngram Viewer (see `https://books.google.com/ngrams`). It allows a user to enter a word and search for historical information on the usage of that word across the years. It also has the capability of searching for multiple words in sequence. For example, you could search for the phrase "use it or lose it", and get information on how often that phrase occurred. For this project, however, we will only be interested in data on individual word usage.

For this project, you will be working with a subset of the database, and doing statistical analysis to glean information from the data. In particular, you will write programs to solve the following tasks:

1. Compute relative letter frequency. In this task, you will compute the relative frequencies of the 26 English letters across all data in our database.
2. Compute the number of printed words. In this task, you will compute the total number of printed words as a function of year. Ultimately you will plot this information and see how the total number of printed words has increased or decreased over the years.
3. Compute trends. In this task, you will choose a starting year and an ending year, and identify which words saw the greatest increase (or decrease) in usage in this timespan.
4. Find most similar words. In this task, you will calculate most similar words using the historical counts of words. The smaller the word count differences between words over the years, the more similar they will be considered.

2 Getting Started

2.1 The Data

Google provides approximately 4 Gigabytes of zip files representing unigram data. In order to make the data size more manageable for this project, we have filtered the data in several ways:

- Only data from the years 1900-2008 have been retained.
- Only words that are entirely lowercase letters have been retained.
- Only words that have appeared at least 10,000 times in print for some year have been retained.

We have further partitioned the data into several files to allow testing with all, or only a portion of the data. The smaller test files allow for faster testing. The complete data file allows for more thorough analysis.

- All words.
- Words beginning with a.
- Words beginning with z.
- Words beginning with a vowel.
- A very short data file containing partial data for only three words.

2.2 Provided Code

In addition to the data files, we are providing (or have already provided) several source files for the project. These include:

- `testLetterFreq.py`: for testing your program for task 1.
- `testPrintedWords.py`: for testing your program for task 2.
- `testTrending.py`: for testing your program for task 3.
- `testWordSimilarity.py`: for testing your program for task 4.

These provided files **should not be changed**. They can be downloaded from:

`http://www.cs.rit.edu/~csci141/Projects/01/unigram.zip`.

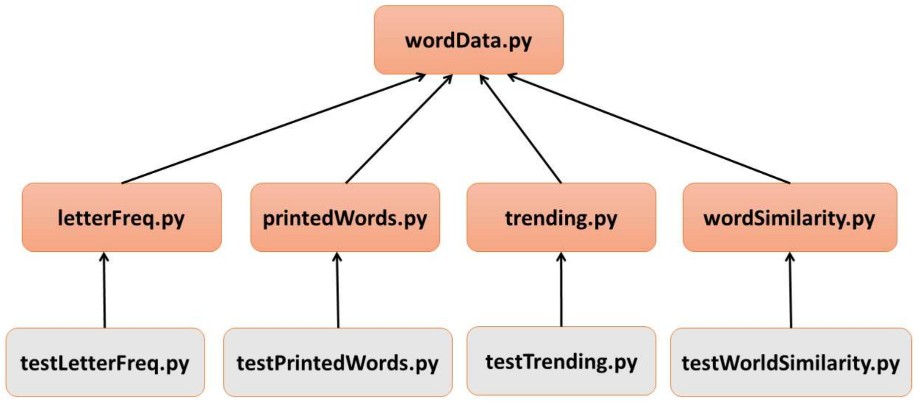
2.3 Code You Will Write

You will write the following programs from scratch:

- `wordData.py`: a supporting module for all tasks. We'll refer to this as "task 0".
- `letterFreq.py`: the main program for task 1.
- `printedWords.py`: the main program for task 2.
- `trending.py`: the main program for task 3.
- `wordSimilarity.py`: the main program for task 4.

2.4 Visual Representation of Project Files

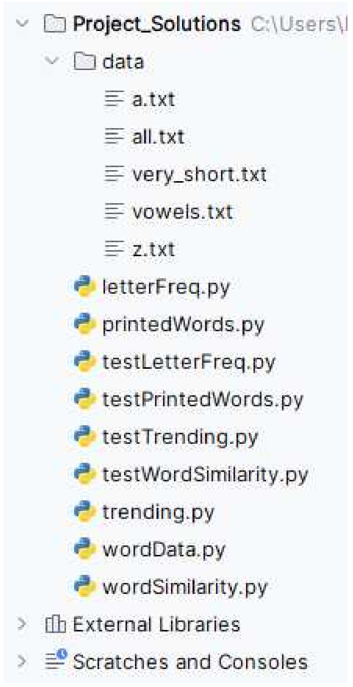
Below is a visual representation of the files involved in this project. Arrows from one file to another indicate that the file at the head of the arrow is imported by the file at the tail of the arrow. Files with a light red background are ones you are responsible for writing.



2.5 Project Structure in PyCharm

You should create a new project in PyCharm using the provided zip file. There should be a `data/` subdirectory with the data files, along with six source files. *DO NOT CHANGE THE CONTENTS OF ANY SUPPLIED FILE!*

The layout of your project should eventually look similar to the following:



```
Project_Solutions C:\Users\
  data
    a.txt
    all.txt
    very_short.txt
    vowels.txt
    z.txt
  letterFreq.py
  printedWords.py
  testLetterFreq.py
  testPrintedWords.py
  testTrending.py
  testWordSimilarity.py
  trending.py
  wordData.py
  wordSimilarity.py
External Libraries
Scratches and Consoles
```

2.6 Standalone Execution versus Importing for Testing

Each of your files should be capable of being run as a standalone program as well as being imported by another program (e.g., a test module).

When your file is run as a standalone program it should cause a `main()` function to be called and executed. When your file is imported by a test module, the `main()` function should not be called and executed. The way to achieve this is with the following program layout:

```
# program imports, classes, and function definitions here...

def main():
    """docstring for your main function here"""
    # main's code body here...

if __name__ == '__main__':
    # run main() only when directly invoking this module
    main()
# end of program file
```

The conditional statement `if __name__ == '__main__'` will only evaluate to `True` when your program is run directly. When the code is imported by another module, this conditional statement will evaluate to `False` and the `main()` function will not be executed.

This organization will allow you to include code in your main function that is only executed when the file is run directly.

3 The Tasks

In the following, note that you must name your files exactly as specified in order for the testing modules to successfully import them. Furthermore, you must follow the naming, parameter and return value specifications provided for required functions. Failure to follow these specifications will again cause the test modules to fail. You may of course add any additional functions that you want to complete your tasks.

3.0 Utility program (Task 0): `wordData.py`

The `wordData.py` file will be your utility file that defines functions used throughout your various tasks.

3.0.1 Required Function Definitions

1. `readWordFile` function
   You must have a function `readWordFile` defined as follows:
   `readWordFile(fileName)`

   Inputs:

   `fileName`: A string giving the name of a unigram data file. The `readWordFile` function assumes that the provided filename belongs to a file in the `data` folder. The function must prepend the string `'data/'` to the given filename before attempting to open it. Thus the `filename` parameter should *not* include `'data/'`, but rather just the name of the file inside the `data` folder.

Output:
A dictionary mapping words to dictionaries. The "inner" dictionary associated with each word will use **years** as keys and **counts** as values.

Example:

```
>>> import wordData
>>> words = wordData.readWordFile('very_short.txt')
>>> print(words)
{'airport': {2007: 175702, 2008: 173294},
'request': {2005: 646179, 2006: 677820, 2007: 697645, 2008: 795265},
'wandered': {2005: 83769, 2006: 87688, 2007: 108634, 2008: 171015}}
```

Look carefully! Getting the format exactly right is very important. For instance, observe that `words` maps the key `'wandered'` to the "inner" dictionary. In this example, the main dictionary contains 3 key-value pairs, where the key is the word and the values are the inner dictionaries. In the inner dictionary, the keys are years and the word counts are the values.

Data File Format:
There are several unigram data files that are provided: `very_short.txt`, `a.txt`, `z.txt`, `vowels.txt` and `all.txt`. You should move progressively to larger files as you test your programs. Each file has the following format:

```
word_1
year,count
year,count
...
year,count
word_2
year,count
year,count
...
year,count
....
word_k
year,count
year,count
...
year,count
```

The first line of the data file contains a word. Following that are some number of lines

of data, each of which contains an integer year, followed by an integer representing the number of times that word appeared in print that year. This repeats for as many years as there are data for that particular word. It then repeats for as many words as there are in the data file.
There are no empty lines in the data files. The year and count fields of a line are separated by a comma.
Hint: Missing year counts can be represented with zero as their counts. This can help with follow-up tasks.

2. `totalOccurrences` function
   You must have a function `totalOccurrences` defined as follows:
   `totalOccurrences(word, words)`

   Inputs:
   `word`: The word for which to calculate the count. Not guaranteed to exist in `words`.
   `words`: A dictionary mapping words to dictionaries with years and counts.

   Output:
   The total number of times that a word has appeared in print.

   Example:

```
>>> import wordData
>>> words = wordData.readWordFile('very_short.txt')
>>> print(wordData.totalOccurrences('wandered',words))
451106
>>> print(wordData.totalOccurrences('quetzalcoatl',words))
0
```

3.0.2 Standalone Execution

When run as a main program, `wordData.py` should prompt the user for an input data file, then prompt for a word to count. It should then output the total occurrences of that word in that data file.

Example:

```
$ python3 wordData.py
Enter word file: very_short.txt
Enter word: airport
Total occurrences of airport: 348996
```

3.0.3 Importing for Testing

There is no specific module provided to test your `wordData.py` functionality. Instead, it will be tested indirectly through the testing of the other individual tasks.

3.1 Task 1: `letterFreq.py`

In this task you will write a program which can compute the relative frequency of English characters occurring in print.

3.1.1 Required Function Definition

You must have a function `letterFreq` defined as follows:

`letterFreq(words)`

Input:

`words`: A dictionary mapping words to dictionaries with years and counts.

Output:

A string containing the 26 lowercase characters in the English alphabet, sorted in decreasing order of frequency of occurrence of each character.

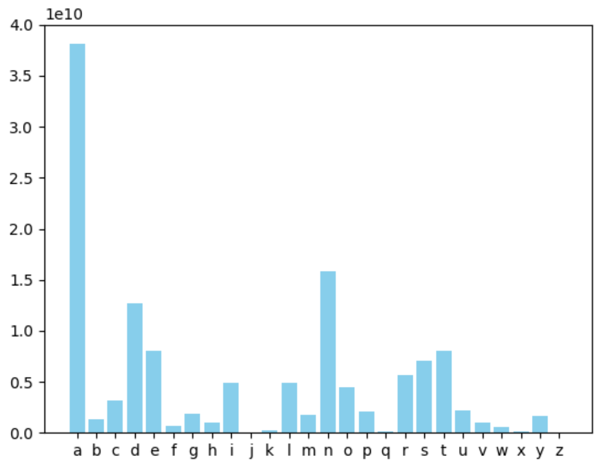
Example:

```
>>> import letterFreq
>>> import wordData
>>> words = wordData.readWordFile('a.txt')
>>> letterFreq.letterFreq(words)
'andetsrliocupgmybhvfwkxqjz'
```

3.1.2 Standalone Execution

When run as a main program `letterFreq.py` should prompt the user for an input data file. It should then output the string containing the 26 lowercase English letters in sorted, decreasing order of frequency. Plot the counts' distribution.

Example:



```
$ python3 letterFreq.py
Enter word file: a.txt
Letters sorted by decreasing frequency: andetsrliocupgmybhvfwkxqjz
```

Important: To plot the character counts, you need to include the following code:

```
import matplotlib.pyplot as plt
<...>
plt.bar(list(letters), list(letter_counts), color='skyblue')
plt.show()
```

3.1.3 Importing for Testing

The test file, `testLetterFreq.py`, has been provided to you to test your implementation of the first task. When you run the `testLetterFreq.py` program, it will import your `letterFreq.py` module and confirm that it correctly computes the output string of letters for the input file `a.txt`. This test file also checks that you are correctly reading the input data file, and that your utility function `totalOccurrences` works correctly.

3.2 Task 2: `printedWords.py`

In this task you will write a program which can compute the total number of printed words for each year.

3.2.1 Required Function Definitions

1. `printedWords` function
   You must have a function `printedWords` defined as follows:

`printedWords(words)`

Input:
`words`: A dictionary mapping words to dictionaries with years and counts.

Output:
A list containing tuples (*year*, *count_total_words*) for each year for which data exists. The list must be sorted in ascending order of year.

2. `wordsForYear` search function
You must have a function `wordsForYear` defined as follows:

`wordsForYear(year, yearList)`

Inputs:
`year`: an integer representing the year being queried.
`yearList`: a list of tuples (*year*, *count_total_words*), as defined in the previous function. The list must be sorted in ascending order of year.

Output:
An integer count representing the total number of printed words for that year. If there is no entry in `yearList` for the requested year, the function returns 0.

3.2.2 Standalone Execution

When run as a main program, `printedWords.py` should prompt the user for an input data file, then prompt the user for a year to count. It should then output the total number of printed words in that year.

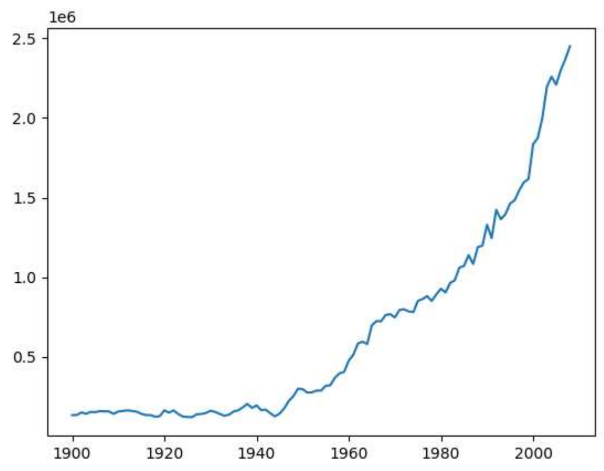
Example:

```
$ python3 printedWords.py
Enter word file: z.txt
Enter year: 1928
Total printed words in 1928 :  143011
```

Your main program should generate a complete plot of the number of printed words as a function of the year. For plotting, you should use the following code:

```
import matplotlib.pyplot as plt
<...>
plt.plot(list_of_years, list_of_counts)
plt.show()
```

The generated plot for `z.txt` should look like the following:



3.2.3 Importing for Testing

The test file, `testPrintedWords.py`, has been provided to you to test your implementation of the second task. When you run the `testPrintedWords.py` program, it will import your `printedWords.py` module and confirm that it correctly computes yearly printed word counts for the data file `z.txt`. The test module does not check the accuracy of every year count; it checks only for a subset of years.

3.3 Task 3: `trending.py`

In this task you will write a program which can compute the top and bottom trending words between a given starting and ending year.

3.3.1 Required Function Definition

You must have a function `trending` defined as follows:

```
trending(words, startYr, endYr)
```

Inputs:

`words`: A dictionary mapping words to dictionaries with years and counts.

`startYr`: An integer, the starting year for the computation of the trending words.

`endYr`: An integer, the ending year for the computation of the trending words.

Output:

A list containing a tuple (`Word, WordTrend`) entry for each word for which qualifying data exists. The list is sorted in decreasing `trend` value.

The `trend` value for each word should be calculated according to the following formula. For each word in the `words` dictionary, a `WordTrend` object is created for that word only if that word has a yearly count of at least 1000 for both the starting year and the ending year. Assuming that the word meets this requirement, the `trend` value for that word is calculated as the ratio of the number of times the word appeared in the ending year divided by the number of times the word appeared in the starting year.

3.3.2 Standalone Execution

When run as a main program, `trending.py` should prompt the user for an input data file, then prompt the user for a starting year, then prompt the user for an ending year. It should then output the top 10 trending words in that timespan. It should then output the bottom 10 trending words in that timespan. The bottom 10 should be output starting from the smallest `trend` value, and finishing with the 10^{th} smallest `trend` value. If the input data file has fewer than 10 words, the output should contain as many words as are available.

Example:

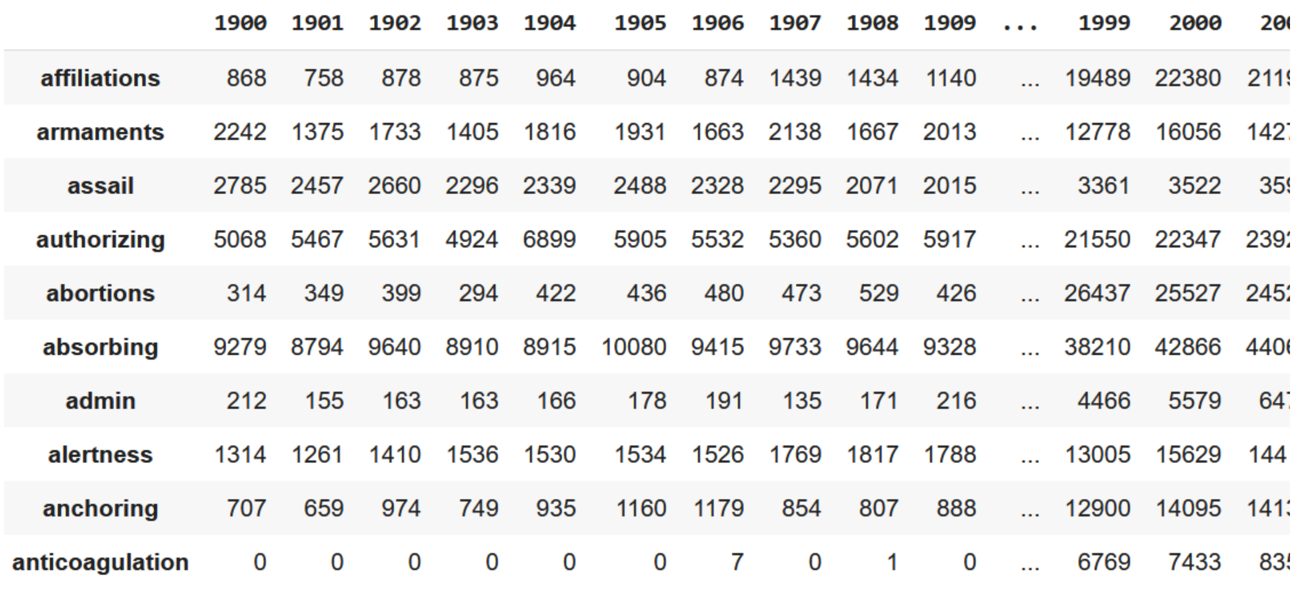
```
$ python3 trending.py
Enter word file: a.txt
Enter starting year: 1966
Enter ending year: 1975
The top 10 trending words from 1966 to 1975 :
algorithms
aversive
amphetamine
algorithm
activism
angiotensin
academics
accountability
assembler
axonal

The bottom 10 trending words from 1966 to 1975 :
adulterated
adj
aeroplanes
anode
aeroplane
archbishop
albumen
armistice
arithmetical
airfields
```

3.3.3 Importing for Testing

The test file, `testTrending.py`, has been provided to you to test your implementation of the third task. When you run the `testTrending.py` program, it will import your `trending.py` module and confirm that it correctly computes trend values for the data file `a.txt` for a variety of different starting and ending years.

3.4 Task 4: `wordSimilarity.py`

For this task, we can think about the data we are working with as a table, where each row is associated with a word, and each column is associated with an year. In this table, the cells could store the counts of a given word in a given year. Missing year counts can be represented with zero as their counts.

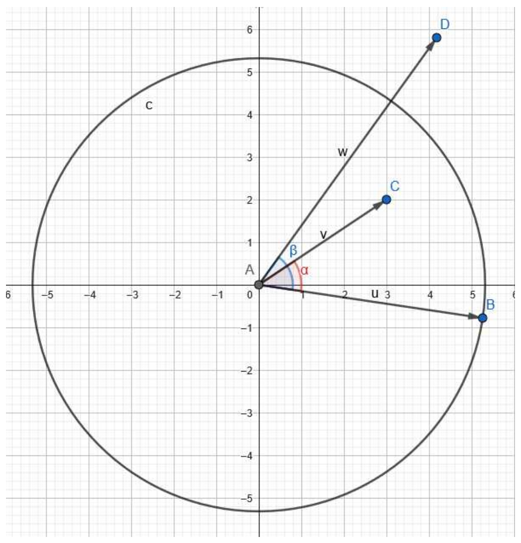


	1900	1901	1902	1903	1904	1905	1906	1907	1908	1909	...	1999	2000	20
affiliations	868	758	878	875	964	904	874	1439	1434	1140	...	19489	22380	211
armaments	2242	1375	1733	1405	1816	1931	1663	2138	1667	2013	...	12778	16056	142
assail	2785	2457	2660	2296	2339	2488	2328	2295	2071	2015	...	3361	3522	35
authorizing	5068	5467	5631	4924	6899	5905	5532	5360	5602	5917	...	21550	22347	239
abortions	314	349	399	294	422	436	480	473	529	426	...	26437	25527	245
absorbing	9279	8794	9640	8910	8915	10080	9415	9733	9644	9328	...	38210	42866	440
admin	212	155	163	163	166	178	191	135	171	216	...	4466	5579	64
alertness	1314	1261	1410	1536	1530	1534	1526	1769	1817	1788	...	13005	15629	144
anchoring	707	659	974	749	935	1160	1179	854	807	888	...	12900	14095	141
anticoagulation	0	0	0	0	0	0	7	0	1	0	...	6769	7433	83

From this table, if we choose a full row associated with a word, we would end up with a series of values, which we could conveniently consider to be a multi-dimensional vector. For curiosity, the number of dimensions in this vector is the number of unique years present in the data. For instance, the word 'admin' would be represented by the following vector:

$$[212, 155, 163, 163, 166, 178, 191, 135, 171, 216, \ldots]$$

From geometry, we know that if two vectors are aligned, the cosine of the angle between them is equal to one, and if they are perpendicular, the cosine is zero. In the figure below, we can see four vectors, namely "B", "C" and "D". Vectors "C" and "D" make the angle β while Vectors "C" and "B" make the angle α.



We can use the value of the cosine of the angle between vectors as the measure of similarity between two words.

The formula for the cosine between two vectors $word_1$ and $word_2$

$$word_1 = [x_1,\ x_2,\ x_3, \ldots,\ x_N]$$

$$word_2 = [y_1,\ y_2,\ y_3, \ldots,\ y_N]$$

can be calculate by the following formula:

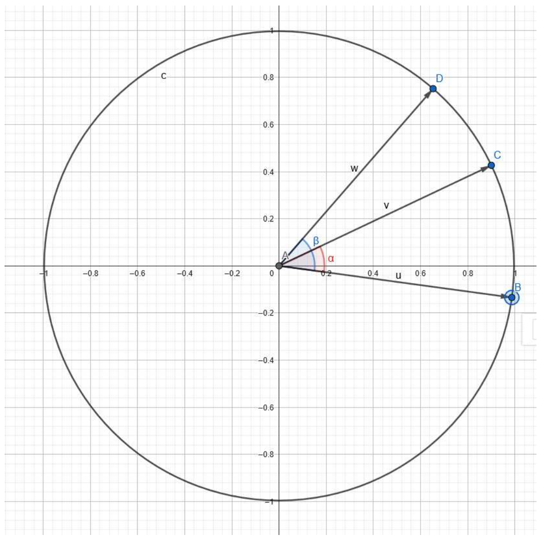
$$\cos(word_1\ word_2) = \frac{x_1 y_1 + x_2 y_2 + \ldots + x_N y_N}{\sqrt{x_1^2 + x_2^2 + \ldots + x_N^2}\sqrt{y_1^2 + y_2^2 + \ldots + y_N^2}}$$

We can then rearrange it as follows:

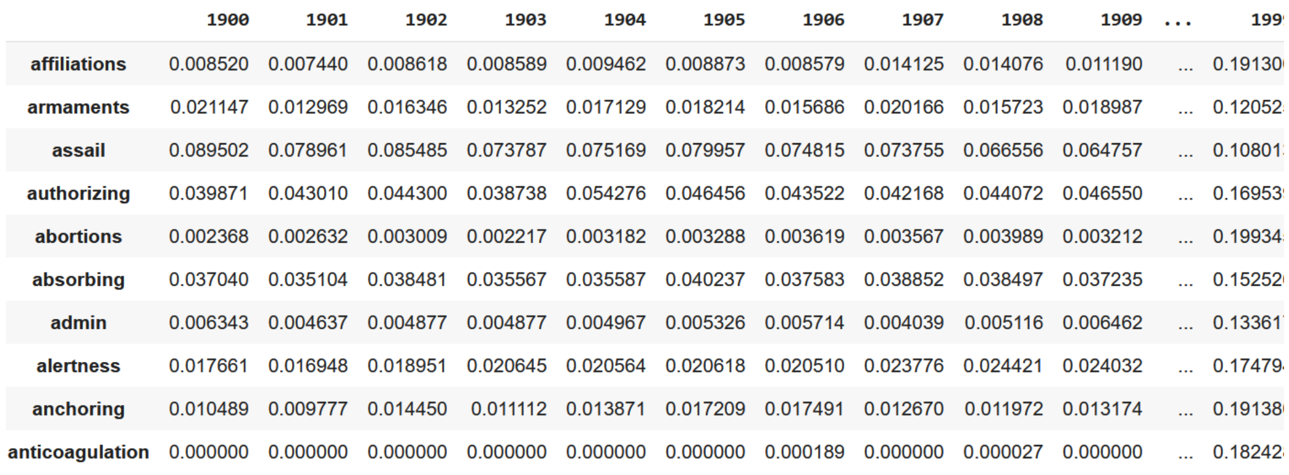
$$\cos(word_1\ word_2) = \frac{x_1}{\sqrt{x_1^2 + \ldots + x_N^2}} \frac{y_1}{\sqrt{y_1^2 + \ldots + y_N^2}} + \ldots + \frac{x_N}{\sqrt{x_1^2 + \ldots + x_N^2}} \frac{y_N}{\sqrt{y_1^2 + \ldots + y_N^2}}$$

Notice that now the denominators are simply the **lengths** of our vectors x and y. Recall that the length of a vector is the square root of the sum of their squared components. For convenience, we can now name our vectors $\tilde{x}$ and $\tilde{y}$ as the **unit vectors** of our vectors x and y, which are simply the original vectors divided by their each own length.

$$\cos(word_1\ word_2) = \tilde{x}_1\tilde{y}_1 + \tilde{x}_2\tilde{y}_2 + \ldots + \tilde{x}_N\tilde{y}_N,$$



To calculate the similarity between a pair of word vectors, we first transform them into unit vectors, as we did in the previous step, by dividing each vector by its length. This process is usually called **normalization**, because all vectors end up with a new “normal” length of 1. After normalization, we would end up with the following updated table.



	1900	1901	1902	1903	1904	1905	1906	1907	1908	1909	...	199
affiliations	0.008520	0.007440	0.008618	0.008589	0.009462	0.008873	0.008579	0.014125	0.014076	0.011190	...	0.19130
armaments	0.021147	0.012969	0.016346	0.013252	0.017129	0.018214	0.015686	0.020166	0.015723	0.018987	...	0.12052
assail	0.089502	0.078961	0.085485	0.073787	0.075169	0.079957	0.074815	0.073755	0.066556	0.064757	...	0.10801
authorizing	0.039871	0.043010	0.044300	0.038738	0.054276	0.046456	0.043522	0.042168	0.044072	0.046550	...	0.16953
abortions	0.002368	0.002632	0.003009	0.002217	0.003182	0.003288	0.003619	0.003567	0.003989	0.003212	...	0.19934
absorbing	0.037040	0.035104	0.038481	0.035567	0.035587	0.040237	0.037583	0.038852	0.038497	0.037235	...	0.15252
admin	0.006343	0.004637	0.004877	0.004877	0.004967	0.005326	0.005714	0.004039	0.005116	0.006462	...	0.13361
alertness	0.017661	0.016948	0.018951	0.020645	0.020564	0.020618	0.020510	0.023776	0.024421	0.024032	...	0.17479
anchoring	0.010489	0.009777	0.014450	0.011112	0.013871	0.017209	0.017491	0.012670	0.011972	0.013174	...	0.19138
anticoagulation	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000189	0.000000	0.000027	0.000000	...	0.18242

Next, we calculate the **dot-product** between the two normalized vectors.

$$dot\_product(\tilde{x}, \tilde{y}) = \tilde{x}_1\tilde{y}_1 + \tilde{x}_2\tilde{y}_2 + \ldots + \tilde{x}_N\tilde{y}_N,$$

If you repeat this process between a word and all others, you will end up with measures of similarity between that word and all other. The greatest values will correspond to the most similar word vectors.

Optional: To compute the similarities between one word and all other words efficiently you may want to use NumPy library. If you have a numpy array `word_counts` of size (number_of_words × number_of_years) and a `word_vector` of size (number_of_years), then `word_counts.dot(word_vector)` will give you the array of size (number_of_words).

3.4.1 Required Function Definition

You must have a function `topSimilar` defined as follows:

`topSimilar(words, word)`

Input:

> `words`: A dictionary mapping words to dictionaries with years and counts.
>
> `word`: a word, for which we are looking for similar words.

Output:

> A list of the top five words including the input word in the descending order of similarity. If there are less than five words in the file, then return as much words as you have. If there is no such a word in the words dictionary, return just one word, itself.

Example:

```
>>> import wordSimilarity
>>> import wordData
>>> words = wordData.readWordFile('all.txt')
>>> word = 'admin'
>>> wordSimilarity.topSimilar(words, word)
['admin', 'authenticate', 'populate', 'hacking', 'deploying']
```

3.4.2 Standalone Execution

When run as a main program `wordSimilarity.py` should prompt the user for a filename and a word. It should then calculate the top-5 most similar word and output them to the terminal.

Example:

```
$ python3 wordSimilarity.py
Enter word file: all.txt
Enter word: admin
The most similar words are:
['admin', 'authenticate', 'populate', 'hacking', 'deploying']
```

3.4.3 Importing for Testing

The test file, `testWordSimilarity.py`, has been provided to you to test your implementation of the forth task. When you run the `testWordSimilarity.py` program, it will import your `wordData.py` module and confirm that it correctly computes the five top similar words for several files in the `data` directory.

4 Testing

Please make sure that your program modules provide the interface required by the test programs. These files have been provided for you to develop and test your modules:

- `testLetterFreq.py`,
- `testPrintedWords.py`,
- `testTrending.py` and
- `testWordSimilarity.py`

Do not change the interface of any of these functions because the test programs will be used to verify your code.

Your program will be tested against different, previously unseen, test cases, which are based on the same data files as those provided.

You should make sure that your code scales well on the large dataset `all_words.txt`. Note that running the test programs on this large data set can take several minutes to finish.

5 Submission

Please submit your code in a file called `abc1234.zip`, where `abc1234` is your RIT username. This file must contain *only* the files highlighted in red in section 2.4.

The zip file structure must be "flat"; that means there must be *no folders within the zip file.* Your zip must not contain any nested folders, nor may any nested folders contain your `.py` files.

Create a separate folder with your username, and then copy only the files you wrote into it, to be zipped up. Upload `abc1234.zip` to the MyCourses Project dropbox.

Do not use any compression mechanism other than zip.

You may submit your project to the dropbox for a "late submission" period of eight hours beyond the posted due date. This incurs a 20% penalty.

6 Grading

6.1 Grading Breakdown

Here is the grading outline for your implementation(100%):

- 10% The functions in `wordData.py`;
- 20% The functions in `letterFreq.py`, including the plot;
- 20% The functions in `printedWords.py`, including the plot;
- 20% The functions in `trending.py`;
- 20% The functions in `wordSimilarity.py`;

- 10% Documentation and Style:
  Each file must have a *header docstring* containing the file name and your name. (Remember, docstrings have 'triple quote' syntax).
  Each function must have its own, *function docstring* with the following:
  - description of input parameters and types; and
  - description of the function's return value(s) and types.

  Further, the code should be well organized on the page. There should be at least one blank line between each component (i.e., between between functions).

6.2 Grading Deductions

It is possible that you got everything working but cheated to succeed, or you failed to follow directions and made testing, cheat-checking and grading of your work very difficult and time-consuming. This leads to these possible deductions over and above the grading breakdown described earlier.

- 100% If **cheat-checking** detects that your modules are strongly similar to those of others, you will be subject to RIT's Academic Honesty Policy as detailed in `http://www.cs.rit.edu/~csci141/syllabus.html`. That means you will receive a 0 grade, and all parties to the cheating incident will receive that treatment. If the infraction is more serious, there may be further penalties (see the syllabus).
- 100% If your modules do not work with the provided test programs, you will receive a 0 grade. If the test programs cannot execute your functions, or if your functions cause the test programs to crash, then you will fail this assignment. You must name the specified functions *exactly as used in the testers*. If you fail to name the functions exactly as stated, you have broken the interface specification. Use the provided testing code to test your code and ensure that you follow the interface with your implementations.
- 5% If you fail to follow the zip submission directions exactly, you will receive a 5% penalty off the top of your earned grade. You must not include anything in your zip file that was not specified in the submission section.
- 10% If you misuse Python dictionaries in a terribly inefficient manner causing your programs to run poorly when fed the `all_words.txt`, you will receive a 10% penalty off the top of your earned grade. Here is an example of what is meant by "terribly inefficient":

  ```
  # look up the word in the dictionary
  for word in words:
      if word == myWord:
          # found it! Now do some more work...
  ```

  Such code indicates a poor understanding of Python's dictionaries!