School of Mathematical & Computer Sciences

Department of Computer Science

Course F21SC: Industrial Programming
"Coursework 2 : Data Analysis of a Document Tracker"

Paparisteidis Georgios - H00202663
gp4@hw.ac.uk

Professor: Hans-Wolfgang Loidl

December 5, 2014
Edinburgh

Table of Contents

1.Introduction

The purpose of this report, is to describe step by step the functionality and the development process of an application that analyses data derived from a document tracker. It offers a guide to the users who want to use the application in order to extract useful information from publications hosted at the popular website http://www.issuu.com, as well as to developers who want to implement a similar project or understand the implementation of this one.

This project was developed using Python 3.4 which is the latest version, using PyCharm IDE that offers a very friendly developing environment under Windows 8. The development and testing of this application offered a great introduction to the Python language and its functionalities. Python is considered a powerful scripting language and can be a great tool for any modern programmer since it offers flexibility and reusability under an object oriented prism, along with loose syntactic rules that make it suitable for rapid prototyping when execution speed is not critical.

This report explains in detail the developing process and the different user scenarios. It was developed as a coursework for F21SC: Industrial Programming at Heriot-Watt University, for the Software Engineering Msc programme supervised by professor Hans-Wolfgang Loidl.

2.Requirement's Checklist

Requirements	Result	Details
1.Python	Fully Completed	Python 3.4 was used for the deployment of the project. The IDE PyCharm was used on Windows 8.1
2.Views by country/continent	Fully Completed	a)Histograms that show the number of **countries** of the viewers is being displayed b)Histograms that display the **continents** of the viewers is being returned. In addition to the histograms both options also return dictionaries with countries/continents and number of occurrences.
3.Views by browser	Fully Completed	a) Histogram with the user agents of the viewers is being returned along with a printed dictionary. b) Histogram with the browser names of the viewers is being returned along with a dictionary.
4.Reader profiles	Fully Completed	A top 10 list is being printed that returns viewers Ids based on the time they have spent reading documents.
5."Also likes" functionality	Fully Completed	“Also like” functionality implemented that returns top 10 document Ids based on : a) The most avid readers that have read these documents b)The number of times a document has been read
6.GUI Usage	Not Completed	The application does not include a GUI
7.Command-line usage	Fully Completed	The application provides a command-line interface for executing different tasks

All of the requirements have been met except for the GUI deployment. The application can run from the command-line and the results are returned on the console windows along with the histogram

3.Design Considerations

The application was designed to offer solutions on analyzing data from a document tracker. The intended users are the ones that want to gain access to some interesting statistics that accompany a specific document. The application offers capabilities such as:

- Analyzing the number of countries and/or continents that a documented has been visited from. (Option 2a/2b)
- The readers that have read a specific document (Option 5a)
- The documents that a specific reader has read (Option 5b)
- Documents also liked by other readers based on their time spent reading (Option 5d)
- Documents also liked by other readers based on the times that have been read (Option 5e)

The application does not include a GUI that would have made the user interaction more friendly but offers a very simple command line usage, where a user can specify a reader ID, a document ID and the task that he wants to execute and the results are being print on screen in a friendly, sorted manner. Furthermore the use of graphical representations (histograms) provides an easy way for depicting data and extracting information from them. The use of these graphs is offered in options 2a,2b,3a,3b.

Overall the user has to choose between the following options mentioned before and are more thoroughly described through the next chapters. It should be mentioned that for a user to make a document or a reader search he has to know the document's or the reader's id, as they are specified in the JSON file and unluckily they do not have a very user friendly format.

4.User Guide

In order to access and run the application a user, using a Linux OS has to navigate himself to the directory where the application files are located. The files needed for the application to execute are :

- cw2.py (which is the main function of the program)
- program.py (which includes all the functions/tasks, along with the libraries)
- issuu_cw2_fixed.json (which is the JSON files that includes all the data)
- countries_continents_mapping.py (which includes two dictionaries that help mapping countries to continents)

In addition to the above the intended user has to have matplot library installed on his/her computer along with a Python 3.4 compiler of course. Matplot libraries can be downloaded from http://matplotlib.org/ and Python3 from https://www.python.org/download/releases/3.0/

Once the user has completed all of the above he is ready to execute the program.

In the example scenario the necessary files have been placed inside the path *~/Documents/Python.* That folder contains the above mentioned files.

```
passas@passasmini:~/Documents/Python $ ls
countries_continents_mapping.py  formatJSON.py        program.py
cw2.py                           issuu_cw2_fixed.json
```

Illustration 1: Linux folder

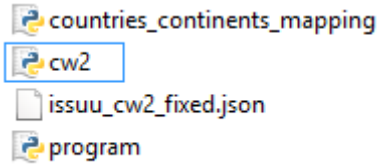

Illustration 2: Windows Folder

The application can be executed via the command line just by typing:

"python3 cw2.py -u [a user's ID] -d [document's id] -t [task number]"

For example if a user wants to search statistics for a document with an ID: 1**40218233015-c848da298ed6d38b98e18a85731a83f4** and he wants to view the countries from which it has been accessed (Task 2a) he has to type:

"python3 cw2.py -d 140218233015-c848da298ed6d38b98e18a85731a83f4 -t 2a"

4.1 Views by country/continent - Option 2a/2b

This option returns a histogram, which is a graphical representation of the countries or continents that a specific document has been accessed from. The user has to specify the document ID along with the option 2a – for countries or 2b – for continents and the histograms along with a printed list are being returned.

Views by country – Option 2a

For example,the document with the ID: **140218233015-c848da298ed6d38b98e18a85731a83f4** will return the following histogram along with a list:

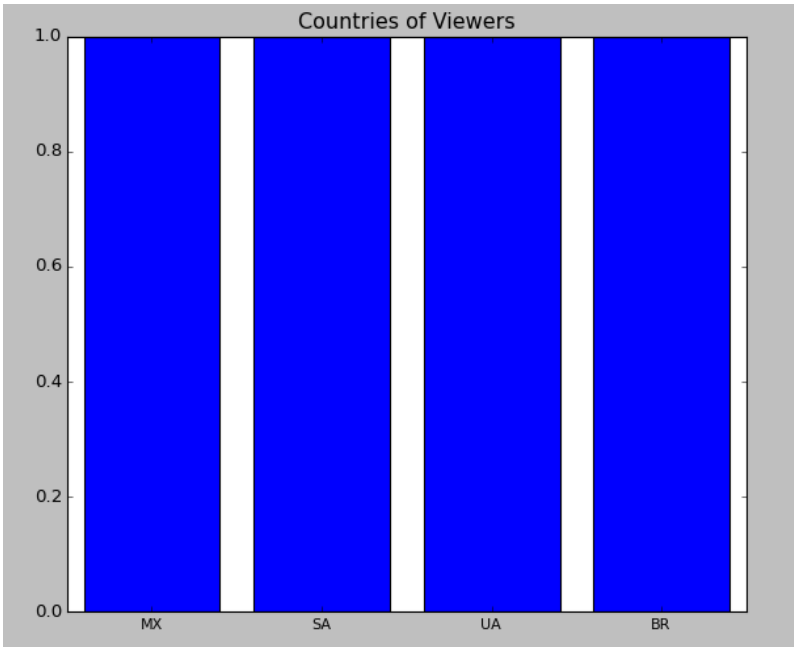

Illustration 3: Histogram of Countries

Countries of Visitors:
MX --> 1
BR --> 1
UA --> 1
SA --> 1

Text 1: Dictionary of countries

It can be seen that this specific document has been visited by 4 countries with the names MX,SA,UA and BR, one team from each.

The xx' axis shows the country codes while yy' axis shows the number of occurrences.

On another example a document with the ID: 140226164301-db02e4587f79095920154d4cb44cec8a returns the following histogram and list:

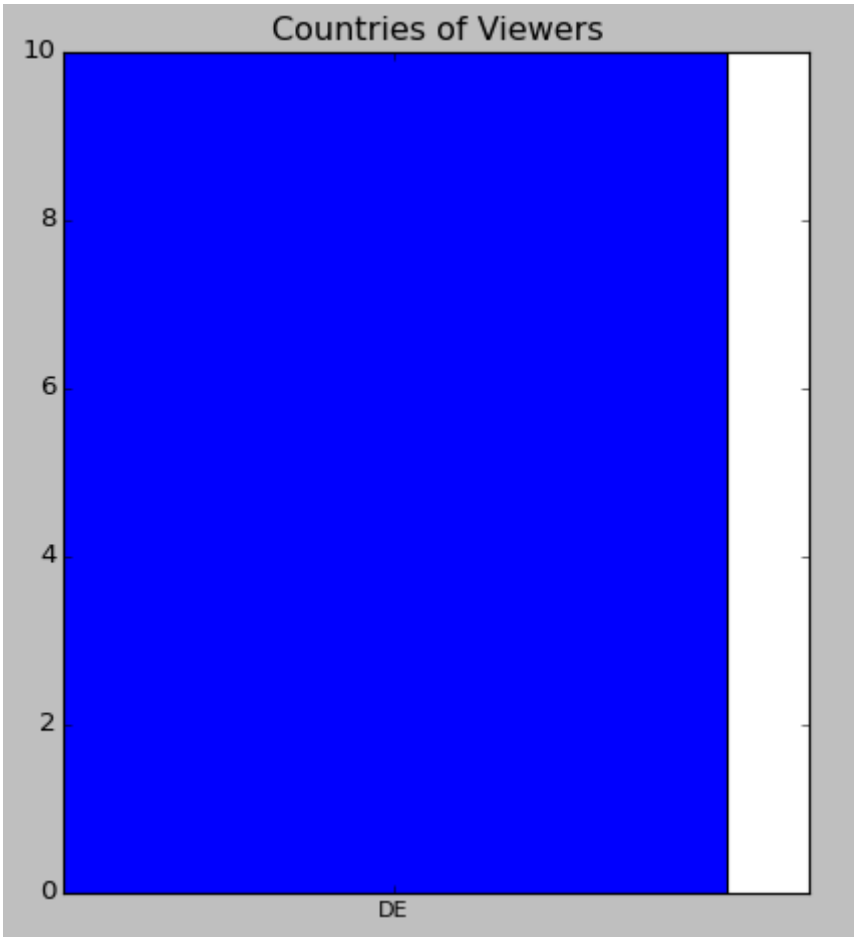

Illustration 4: Histogram of Countries

Continents of Visitors:

South America --> 1

Asia --> 1

Europe --> 1

North America --> 1

Text 2: Dictionary of countries

In this example, the specific document has only been accessed from one country, Denmark (DE), but it has been accessed 10 times.

Views by continents – Option 2b

In addition to the countries functionality, a specific document can be analyzed in regards to the number of different continents that has been accessed from. So for example in the two scenarios above we would expect the first document to return 4 different continents, since these countries belong to four different ones (North America,Europe,South America and Asia) while in the second scenario the only continent that should be returned should be Europe.

The user has to select task 2b in order to see the continents. The specific command for getting the continents of the viewers for document in the first scenario, should be:

"python3 cw2.py -d 140218233015-c848da298ed6d38b98e18a85731a83f4 -t 2b"

In fact the expected histogram and list are being returned, which show that the specific document has been accessed once from each of South America, Asia, Europe and North America

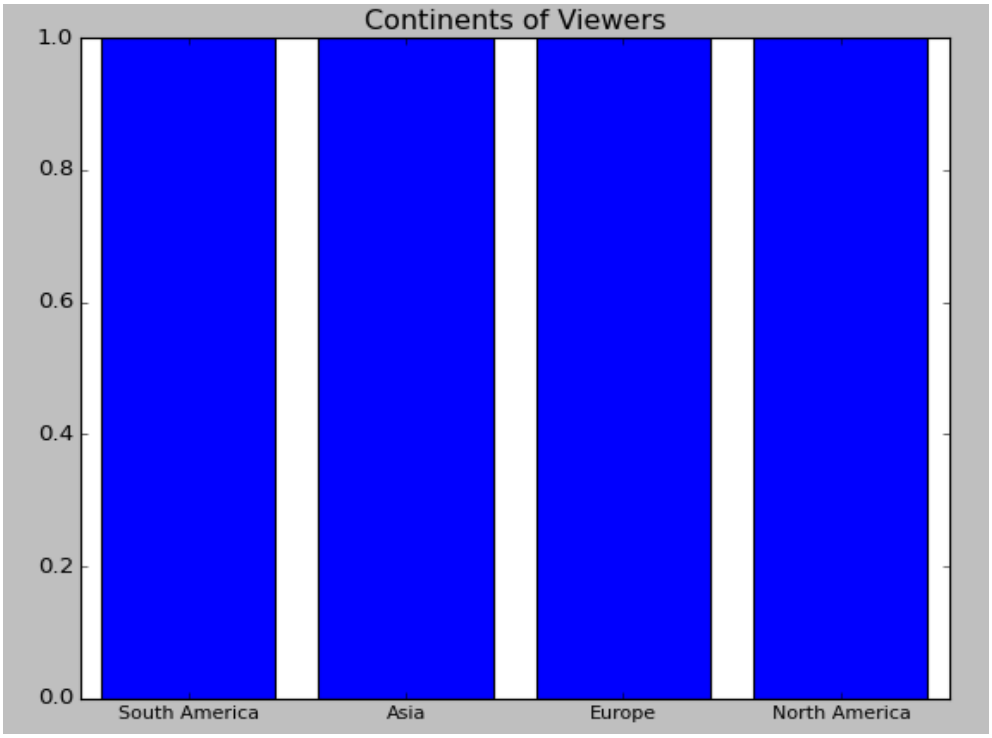

Illustration 5: Histogram of Continents

Continents of Visitors:

North America --> 1

Asia --> 1

South America --> 1

Europe --> 1

Text 3: Dictionary of Continents

4.2 Views by browser – Option 3a/3b

In this option, the user can get information regarding on the browser agents and browser names of all the visitors. The results will again be displayed both in the form of a histogram and in a list. The difference between option 3a and option 3b is that, **3a** returns the full agent names that can be hard to read but give a view of the great variety between different browsers distributions and versions. **3b** on the other hands simplifies the previous results and returns information based on the browser name. More specifically the browsers that are being searched for are : Mozilla Firefox, Google Chrome, Safari, Opera and Internet Explorer. All other browsers go into the category "Others" .

Views by browser identifiers – Option 3a

Using the document's ID from the first scenario (*p.6)* the command to be executed is:

"python3 cw2.py -d 140218233015-c848da298ed6d38b98e18a85731a83f4 -t 3a"

this will return the histogram and the list listed bellow:

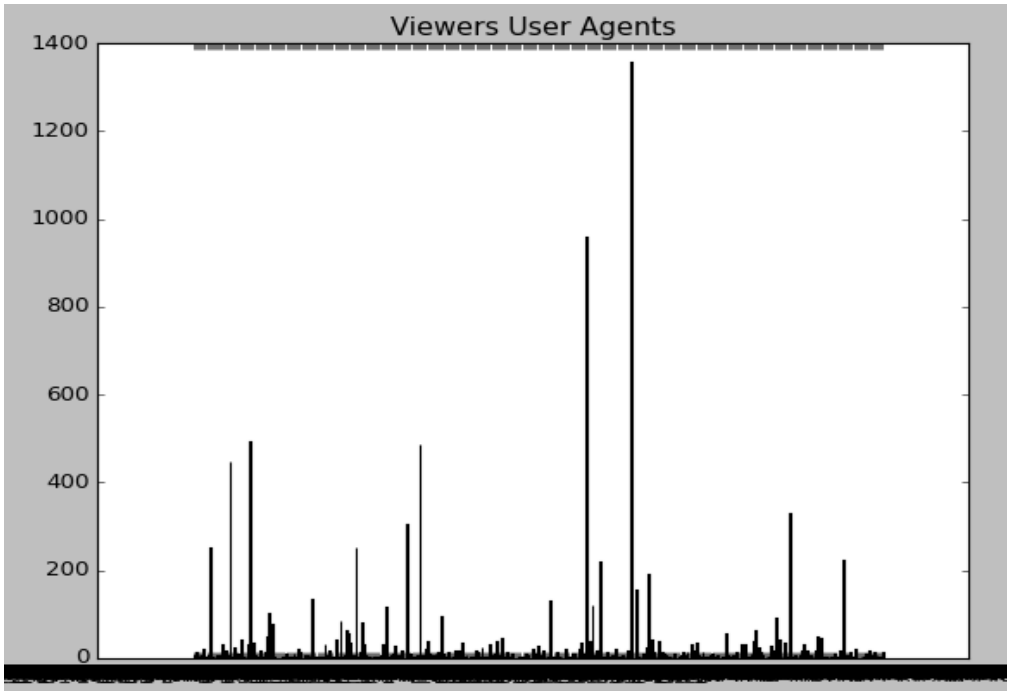

Illustration 6: Histogram Browser Identifiers

Browsers identifiers

Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; MAARJS; rv:11.0) like Gecko --> 22

Mozilla/5.0 (Unknown; Linux i686) AppleWebKit/534.34 (KHTML, like Gecko) Qt/4.7.4 Safari/534.34 --> 4

Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; InfoPath.2) --> 2

Mozilla/5.0 (Windows NT 6.1; WOW64; rv:26.0) Gecko/20100101 Firefox/26.0 --> 18

Text 4: Part of Dictionary of Browser Identifiers

Views by browser names – Option 3b

In a similar way like before, when a user specifies **task 3b** he will get a histogram and a list of the browser names that have accessed all the documents.

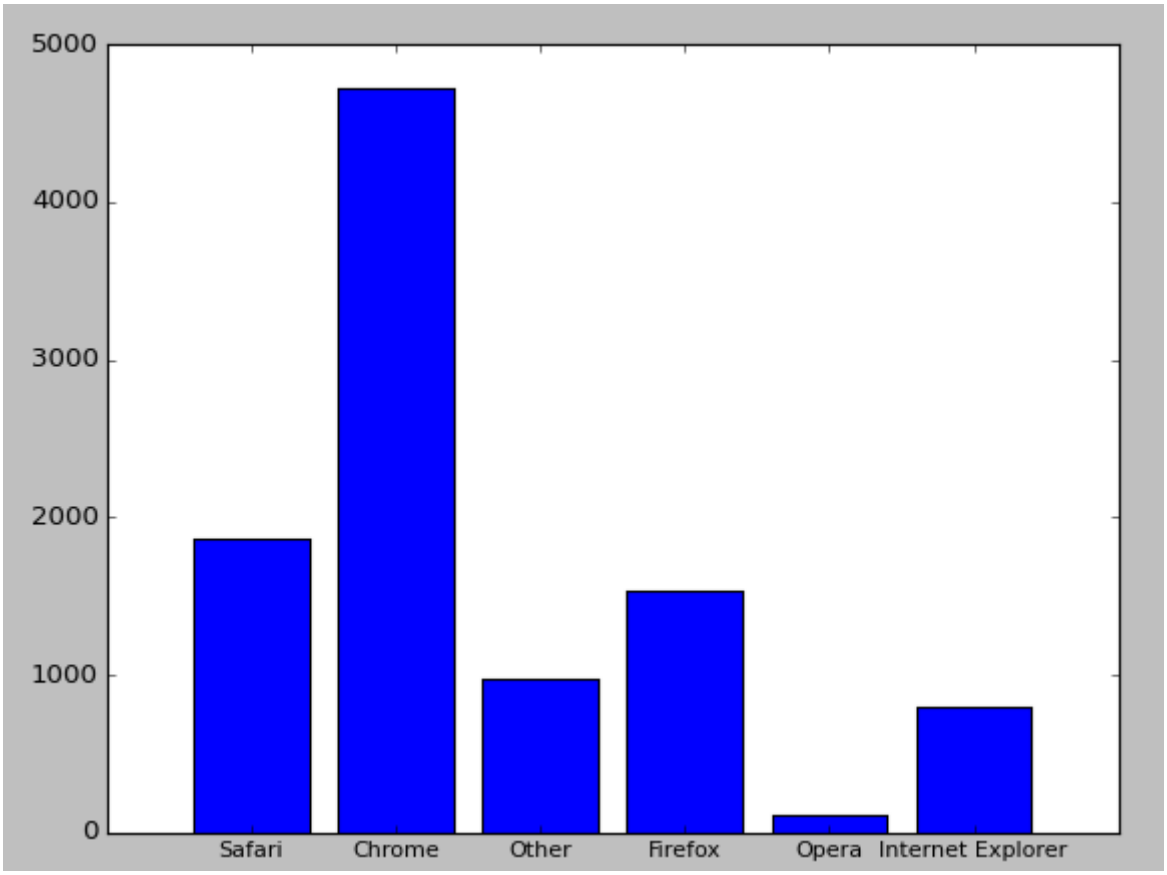

Illustration 7: Histogram of browser names

```
Browsers :
Other --> 977
Chrome --> 4720
Internet Explorer --> 796
Opera --> 107
Firefox --> 1534
Safari --> 1869
```

Text 5: Dictionary of browsers

The above information is very useful since we can see the popularity among browsers and which one is being used more to read documents.

4.3 Reader profiles – Option 4

When selecting option 4, the user can get a top 10 list of the most avid readers. That is, in other words, the readers that have spent the most time reading documents. This function does not require a specific document or a reader ID and can be executed just by specifying the number of the task.

"python3 cw2.py -t 4"

The result as expected is a list with the top 10 reader Ids.

Top Readers are:
e529f034d3430af2
dd326898d5605e63
458999cbf4307f34
849bb060cb110347
df70cddbd46fd5da
14e1e343078d3d75
2105dd9bc68afb9d
da0df8a63107e139
b9caded38e707eca
e1178362fc11d6ba

Text 6: List of top 10 avid readers

4.4 "Also likes" functionality – Option 5a/5b/5d/5e

The also likes functionality provides the user with the capability of finding documents similar to his interests based on what common interests between him and other users. More specifically the user can specify a document ID and then get a list of readers who have read that exact document. Furthermore, given a reader ID he can see what other documents that specific reader has read. Then he can get some recommendations which can be based A) on reader's credibility, that is the time that each reader has spent reading documents or B)the number of times a document has been read.

Readers of a specific document – Option 5a

The user has to specify a document ID and he will get a list of reader Ids that have read that document.

Readers that have read this Document:

76175bb1ea9805a1

cee42a0927c5f2da

232eeca785873d35

489c02f3e258c199

Text 7: Readers who have read this document

Documents of a specific reader – Option 5b

Given a reader's ID the user can have a list of all the documents that have been read by that reader. So for example the reader with an ID : febce31f23dc35eb

"python3 cw2.py -u febce31f23dc35eb -t 4"

will return:

Documents that have been read by this Reader:

140217071905-b3b7d787c2eef33b3d42d8e014b606d2

Text 8: Documents read by a reader

"Also like" documents based on readership profile – Option 5d

This option allows the user to get a list of top 10 documents that are also liked by other readers. This means that given a document 1 , the result will be some other documents that have been read by readers who have also read document 1. The criteria for sorting these documents will be each reader's reading profile. So for example if a reader has spent 200 minutes reading some documents and another has spent 100 minutes, then the documents of the first reader will come on top of the list. Given the document ID in the first scenario (*p.6),* the result will be:

Documents also liked by other readers based on their reading profile:

140218233015-c848da298ed6d38b98e18a85731a83f4

131202094202-a4ae3185bc84368f14bff266d276eb4b

131218101426-7fe24377c762b8fe53d21b65fcfa9b25

100713205147-2ee05a98f1794324952eea5ca678c026

140101075322-b9180e4eddbece0371da647a6ca0e939

130313161023-ee03f65a89c7406fa097abe281341b42

111114223935-a39e830a44fa40099a28f587673c4663

121227235605-3168dae6b46e4593bf1d3a5b18483291

140129035743-7c00ce6256d9fd9f78ef6d1cac869d60

140228142350-9f269f7cc77eed6045f5930e276d280d

Text 9: Suggested documents based on reader profiles

“Also like” documents based readers count – Option 5e

This option returns again a top 10 list of suggested documents that have been read by readers that have originally read the specified document, but in this case the sorting of the top documents is taking place, taking into consideration the number of times that each document has been read. So for example if document 1 has been read by 20 readers and another document 2 has been read by 5, then document 1 will be on top of that returned list.

Using the same document ID as an example, the list returned will be:

Documents also liked by other readers based on popularity:
131202094202-a4ae3185bc84368f14bff266d276eb4b
140218233015-c848da298ed6d38b98e18a85731a83f4
100713205147-2ee05a98f1794324952eea5ca678c026
131218101426-7fe24377c762b8fe53d21b65fcfa9b25
140228083520-000000008d3679dbb78286526bd8c14b
140227101855-42de650464f91d12c6a5644f999c6287
130810070956-4f21f422b9c8a4ffd5f62fdadf1dbee8
140101075322-b9180e4eddbece0371da647a6ca0e939
131105193559-dbac395e3cc43fc2b0077eaf789183bb
130630171409-40897a57ebbe917e46a0735727ae8945

5.Developer Guide

The application is separated over 2 python files. **program.py** that contains the functions for each of the required tasks along with the libraries and **cw2.py** that receives user input and calls the functions to be executed. During the whole implementation specific coding standards tried to be maintained such as declaring all functions with an uppercase first letter, while variables were declared with a lower case one for accessibility. Furthermore lists and dictionaries created had the name of the function that were operating along with the word Dict or List at the end.

5.1 Formatting the JSON file

The purpose of this application is to take data from a given JSON file, then analyze them and do the required tasks, for different user cases. The JSON file was provided and was originally titled issuu_cw2.json and it is provided along with the application code. The problem was that the format of that JSON file was not recognized by python3 and the original file had to be fixed, in order for the application to access it.

More specifically the format of the original JSON file was:

{object1}{object2}{object3}....{object(n-1)} for 10.003 entries as can be seen bellow


```
{"ts":1393631983,"visitor_uuid":"04daa9ed9dde73d3","visitor_source":"external",
"visitor_device":"browser",
"visitor_useragent":"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/33.0.1750.1
"visitor_ip":"6a3273d508a9de04","visitor_country":"ES","visitor_referrer":"64f729926497515c","env_type":"reader",
"env_doc_id":"140224195414-e5a9acedd5eb6631bb6b39422fba6798","event_type":"impression","subject_type":"doc",
"subject_doc_id":"140224195414-e5a9acedd5eb6631bb6b39422fba6798","subject_page":0,"cause_type":"impression"}
{"ts":1393631983,"visitor_uuid":"04daa9ed9dde73d3","visitor_source":"external","visitor_device":"browser",
"visitor_useragent":"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/33.0.1750.1
"visitor_ip":"6a3273d508a9de04","visitor_country":"ES","visitor_referrer":"64f729926497515c","env_type":"reader",
"env_doc_id":"140224195414-e5a9acedd5eb6631bb6b39422fba6798","event_type":"impression","subject_type":"doc",
"subject_doc_id":"140224195414-e5a9acedd5eb6631bb6b39422fba6798","subject_page":1,"cause_type":"page"}
```


While the required formatting should be : [{object1},{object2},.....,{object(n-1)}]

To achieve this a small python script was created that would simply insert a “,” after each object. That script would iterate over the original json file (issuu_cw2.json) and it would replace “}” with “},”. Finally an ending and closing bracket were added manually to the JSON. That small script can be seen bellow, the input was the original file (issuu_cw2.json) and the output was the new formatted JSON (issuu_cw2_fixed.json) that was eventually used for the rest of the application.

```
__author__ = 'Paparisteidis Georgios'

fileIN = "issuu_cw2.json"
fileOUT = "issuu_cw2_fixed.json"
#read old file and store its contents
f = open(fileIN,'r')
data = f.read()
f.close()
#Fix formatting
newData = data.replace("}","},")
#Write to new file
f = open(fileOUT,'w')
f.write(newData)
print("JSON file has been formatted")
```

Illustration 8: Formatting the JSON

5.2 Implementing Functions

All the program's functions are located inside **program.py** and are being called by the **cw2.py** along with user specified arguments such as document ID ,reader ID and task number.

The libraries that were used are: matplotlib, collections and the countries_continents_mapping.py and are all located inside **program.py**

5.2.1 Views by country/continent - *ViewByCountry(docID,user_selection)*

For this and every other function iteration though the fixed JSON file is required and was required. After finding the document with the specified ID, that is being passed on by the user, the user's country, which is the value of the element *visitor_country* is being passed on a list that will contain all the country names under the name *docCountryList*

```
def ViewByCountry(docID,user_selection):
    docCountryList=[]
    for x in jfile:
        if x.get('subject_doc_id') == docID:
            docCountryList.append(x['visitor_country'])
```

Code 1: Countries List

Views by country

Taken the list that was created above, we make use of the function *Counter,* that is included inside the library *collections*. Counter function, counts the occurrences of each element inside a list and maps them to a dictionary. So the dictionary will have the form of :

BR 1
MX 1
UA 1
SA 1

These values are being passed on, two lists that will be the x and y axis of the histogram that will be formed.

```
if user_selection == '2a':
    #Run histogram and return countries list
    x = []
    y = []
    #Insert countries and number of occurences in two seperate lists
    for k,v in Counter(docCountryList).items():
        x.append(k)
        y.append(v)
    plt.title('Countries of Viewers')
    plt.bar(range(len(y)), y, align='center')
    plt.xticks(range(len(y)), x, size='small')
    plt.show()
    return docCountryList
```

Code 2: Countries histogram and list creation

Views by continents

Using again the docCountryList that was created, along with the **countries_continents_mapping.py** file, we are mapping the countries inside the list to specific continents. To do so we iterate each every country inside the list, then find a match in cntry_to_cont dictionary which maps countries to continents and finally we map each continent with each full name (SA → South America).Once the mapping is complete the continent is appended inside the *continentsList.*

```
continents = {
  'AF' : 'Africa',
  'AS' : 'Asia',
  'EU' : 'Europe',
  'NA' : 'North America',
  'SA' : 'South America',
  'OC' : 'Oceania',
  'AN' : 'Antarctica'
}


cntry_to_cont = {
  'AF' : 'AS',
  'AX' : 'EU',
  'AL' : 'EU',
  'DZ' : 'AF',
```

Code 3: Dictionaries for mapping countries and continents

```
elif user_selection == '2b':
    continentsList = []
    #Itterate through countries
    for c in docCountryList:
        for country, continent in cntry_to_cont.items():
            if c == country:
                for contnt,cntntName in continents.items():
                    if contnt == continent:
                        continentsList.append(cntntName)
```

Code 4: Inserting continent in the list

On a similar way like we did with countries and with the use of *Counter* function we iterate through the list and create a dictionary of the continents and occurrences. After that, two lists are created that will include continents' names and continents' occurrences and will form the x and y axis for the histogram.

5.2.2 Views by browser - *ViewsByBrowser(task_selection)*

This function was implemented on a similar way as the previous one. An iteration through each object of the JSON file is required, and the value of the field *visitor_useragent* was extracted and added to the *userAgentList.* Then using the *Counter* function again, a dictionary was created and the two lists extracted from its keys and values were used as axis for the histograms.

For the second part, where the browser names had to be displayed and not the whole browser identifiers, we were searching for keywords inside each *visitor_useragent* element. These keywords were the names of the popular browsers (Firefox,Chrome etc.). If a match was found then the name of the browser was being added to the *browsersList.*

```
if task_selection == '3b':
    browsersList = []
    for x in jfile:
        if "Firefox" in x['visitor_useragent']:
           browsersList.append("Firefox")
        elif "Chrome" in x['visitor_useragent']:
            browsersList.append("Chrome")
        elif "Safari" in x['visitor_useragent']:
            browsersList.append("Safari")
        elif "Opera" in x['visitor_useragent']:
            browsersList.append("Opera")
        elif "MSIE" in x['visitor_useragent']:
            browsersList.append("Internet Explorer")
        else:
            browsersList.append("Other")
```

Code 5: Adding browser names to list

After the creation of the list, the histogram was created using the same logic as before.

```
#Run histogram and return browsersList
x = []
y = []
#Insert countries and number of occurrences in two separate lists
for k,v in Counter(browsersList).items():
    x.append(k)
    y.append(v)
plt.bar(range(len(y)), y, align='center')
plt.xticks(range(len(y)), x, size='small')
plt.show()
return browsersList
```

Code 6: Browsers Histogram

5.2.3 Reader profiles - *TopReaders()*

For this function the top readers had to be calculated, based on their total reading times. To calculate the reading times what was done, was to iterate through the json file and look at specific objects, where the value of the element *event_type* is *pagereadtime.*

```
#Look for page read occurrences
if x['event_type'] == 'pagereadtime':
```

Code 7: Conditional check

Once that check has been made, the visitors_id and his pagereadtime are being added inside *readTimeDict,* which is a dictionary in the form of *realTimeDict[visitor_uuid] = pagereadtime.*.

Since the total amount of readtime for each visitor has to be calculated and some users appear more than once inside the JSON file, there has to be an iteration through the json file in order to check if a user has already been added and add the new reading time to the previous one. This is being accomplished in the following lines of code.

```
131     #Itterate through JSON File
132     for x in jfile:
133         userFound = False
134         #Look for page read occurrences
135         if x['event_type'] == 'pagereadtime':
136             #search for duplicate userID inside dict
137             for k,v in readTimeDict .items():
138                 if k == x['visitor_uuid'] and x['event_readtime'] != None:
139                     #Increment existing readtime value
140                     readTimeDict[k] = int(x['event_readtime']) + v
141                     userFound = True
142                     break;
143             #Add new user entry to dictionary
144             if userFound == False:
145                 readTimeDict[x['visitor_uuid']] = int(x['event_readtime'])
```

A flag *userFound* is being used, in order to guarantee the record of the first element to the dictionary, since the loop cannot be executed in an empty one. Furthermore an extra check is being done in line 138 to secure that the ['event_readtime'] element is not empty.

Once the dictionary is done it is being returned to the main program, for sorting and printing. The reason why the sorting is not being executed inside this function is because, *readTimeDict* is being used by another function (5d) in later stages.

The operations of sorting and printing are being executed in **cw2.py** which takes the dictionary returned from AvidReaders(), where in line 57 the dictionary is being sorted based on the values, in reverse order and the top 10 readers are added inside *topReadersList.*

```
54      topReadersList = []
55      readTimeDict = program.TopReaders()
56      #Sort dictionary based on readTime values,reverse it,add it on list
57      for k,v in (sorted(readTimeDict.items(), key=lambda x:-x[1]))[:10]:
58          topReadersList.append(k)
```

5.2.4 "Also likes" functionality – *AlsoLike(docID,task_selection)*

This function has to return "also liked" documents for a given document ID based on two different criteria. The first one is the readership profile and the second is the documents' read count, meaning by how many different readers each document has been read.

Given a document ID an iteration through the file is being made to check all the readers that have also read this document, and the results are being inserted into *readersList.*

```
def DocToReaders(docID):
    readersList = []
    for x in jfile:
        #Search for doc in JSON,if the element doc_id exists
        if x.get('subject_doc_id') == docID:
            #Insert visitor's ID inside list
            readersList.append(x['visitor_uuid'])
    #return distinct readers IDs
    return set(readersList)
```

Code 8: Readers List based on a document ID

On a similar way and given a reader's ID, all the documents that have been read by that specific reader are being added into *docList.*

```
#5b. Returns DocumentIDs based on a readerID
def ReadersToDoc(readerID):
    docList = []
    for x in jfile:
        #Find user ID
        if x['visitor_uuid'] == readerID:
            #Check that doc ID exists
            if x.get('subject_doc_id') != None:
                #Add document ID to the list
                docList.append(x['subject_doc_id'])
    #return distinct values for document IDs
    return set(docList)
```

Code 9: Documents List based on a reader ID

After these two lists have been filled with readers and documents respectively, a dictionary is created, named *readersDocDict.* This dictionary will map the readersIDs with DocumentIDS.

So for each reader key there will be all the documents that he has read as values. Taking the form bellow.

Reader1	[Doc2,Doc3,Doc15,Doc6]
Reader2	[Doc1,Doc20,Doc15,Doc7,Doc60,Doc12]
Reader3	[Doc120]

Table 1: Readers to Documents Dictionary Example

```
174    def AlsoLike(docID,task_selection):
175        readersDocDict = dict()
176        #Itterate over readers IDs
177        for k in DocToReaders(docID):
178            #Itterate over Document IDs that each reader has read
179            for v in ReadersToDoc(k):
180                #Insert readerID,docID into Dictionary
181                readersDocDict.setdefault(k,[]).append(v)
```

Code 10: Readers to Documents Dictionary

"Also like" - based on readership profile - 5d

Once this dictionary is completed, we are making use of the *readTimeDict* that was created at 5.3.2 for reader profiles, in order to map Reading Times of the users to the documents that they have read.

An iteration through each of those dictionaries is performed and these two dictionaries are being merge into a new one which will include reading times of the users as keys and the documents that they have read as values. In order to make this procedure more clear to understand we are using the diagram bellow to depict these stages.

ReaderID	ReadTime
232eeca785873d35	1100
c08fc48b49f0e1be	520
0826ad759b5d254e	2500

Table 2: readTimeDict

ReaderID	DocumentsID
c08fc48b49f0e1be	[110322220408-aadc46d780e4a7605], [140206010823-b14c9d93a04234af7], [1107-000009cca70787e5fba1fda005c85]
232eeca785873d35	[140206010823-b14c9d966be95031], [000e976612072abfdd0e95]
0826ad759b5d254e	[130701025930-558b150c485fc89], [180e4eddbece0371da647a6ca0e],[3306-ba4f086f3bc24fdd93],[31120234743-616166d17],[81ae8f2b9acdf0324d892]

Table 3: readersDocDict

ReadTime	DocumentsID
1100	[140206010823-b14c9d966be95031], [000e976612072abfdd0e95]
520	[110322220408-aadc46d780e4a7605], [140206010823-b14c9d93a04234af7],[1107-000009cca70787e5fba1fda005c85]
2500	[130701025930-558b150c485fc89], [180e4eddbece0371da647a6ca0e],[3306-ba4f086f3bc24fdd93],[31120234743-616166d17],[81ae8f2b9acdf0324d892]

Table 4: DocReadTimeDict

The new dictionary will contain Reading times mapped to Document Ids. Each user who is represented by his reading time, can have multiple documents.

```
183         #Option 5d for returning a list of documents based on readership profile
184         if task_selection == '5d':
185             #Create dictionary to store DocumentIDs --> Reader's total read time
186             docReadTimeDict = dict()
187             #Itterating through Doc -->
188             for k,v in TopReaders().items():
189                 for key,value in readersDocDict.items():
190                     if k == key :
191                         #Itterate through values
192                         for i in value:
193                             #Insert DocID -- > readTime into dictionary
194                             docReadTimeDict[i] = v
```

Code 11: Creating docReadTimeDict

Once the dictionary is created we can then sort it based on reading times and then get the documents for each of the top readers and add them on a list. The list used for this purpose is called *alsoLikeList* and it includes the top 10 documents based on reading times of the readers. If the top reader has read 6 documents, then these 6 documents will be at the top of the list. The the second reader's documents will be added etc.

```
195             alsoLikeList = []
196             #Itterate through dictionary and sort values based on reader's time.
197             for k,v in sorted(docReadTimeDict.items(), key = lambda x:-x[1])[:10]:
198                 #Insert top 10 values on list
199                 alsoLikeList.append(k)
200             return alsoLikeList
```

Code 12: Also Like list - Based on readership profile

"Also like" - based on number of readers of the same document - 5e

This function has to return suggested documents, for a given document ID which will be based on the number of readers that have read them. The implementation, is quite simple and similar to the above. An iteration through *readersDocDict* is being made, which includes readerIDs and all of the documents that each one has read, and inserts each document inside *docList.* After that each element inside the list is being counted and finally the elements top 10 elements(documents) with the most occurrences are being returned.

```
204             docList = []
205             for k,v in readersDocDict.items():
206                 #Store each document read, inside a list
207                 for i in v:
208                     docList.append(i)
209             countDocDict = dict()
210             #Create dictionary with DocumentsRead --> Number of occurrences
211             for x in docList:
212                 if x in countDocDict:
213                     countDocDict[x] += 1
214                 else:
215                     countDocDict[x] = 1
216             alsoLikeList = []
217             #Insert top 10 documents, based on the times they have been read inside a list
218             for k,v in sorted(countDocDict.items(), key = lambda x:-x[1])[:10]:
219                 alsoLikeList.append(k)
220             return alsoLikeList
```

Code 13: Also Like list - Based on amount of readers

5.3 Main

In order for the application to run from the command line and receive document Ids, readers Ids and the task Id as arguments, *OptParser* from library *optparse* is being used.

```
parser = OptionParser()
parser.add_option('-u', action="store")
parser.add_option('-d', action="store")
parser.add_option('-t', action="store")
options, args = parser.parse_args()

readerID = str(options.u)
docID = str(options.d)
task_selection = str(options.t)
```

Code 14: Parsing inputs

What follows -u will be the readers ID, -d will be document's Id and -t will be task's Id. After the inputs have been parse they are assigned to their respective variables and the program continues its execution. Right after that, and depending on the user's selection the selected task is being used.

In every case the matching list is being returned, iterated and every element of it is being printed.

```
if task_selection == '2a':
    print("Countries of Visitors:")
    for k,v in (Counter(program.ViewByCountry(docID,task_selection))).items():
        print(k,"-->",v)
elif task_selection == '2b':
    print("Continents of Visitors:")
    for k,v in (Counter(program.ViewByCountry(docID,task_selection))).items():
        print (k,"-->",v)
```

```
elif task_selection == '4':
    print("Top Readers are:")
    topReadersList = []
    readTimeDict = program.TopReaders()
    #Sort dictionary based on readTime values,reverse it,add it on list
    for k,v in (sorted(readTimeDict.items(), key=lambda x:-x[1]))[:10]:
        topReadersList.append(k)
    #Return a list representation of the sorted dictionary
    for x in topReadersList:
        print(x)
```

```
elif task_selection == '5d':
    print("Documents also liked by other readers based on their reading profile: ")
    program.AlsoLike(docID,task_selection)
    list = program.AlsoLike(docID,task_selection)
    for x in list[:10]:
        print(x)
elif task_selection == '5e':
    print("Documents also liked by other readers based on popularity (times read): ")
    program.AlsoLike(docID,task_selection)
    list = program.AlsoLike(docID,task_selection)
    for x in list[:10]:
        print(x)
```

Code 15: cw2.py

6.Testing

Several different cases were tested, to verify that the application does not crush and follows the expected behavior.

Case Tested	**Expected Result**	**Actual Result**
Executing with no arguments	Do nothing – then exit	The expected
Execute task 5b with no user ID	Print empty list	The expected
Execute task with no doc ID	Print empty list/Empty histogram	The expected
Corrupt JSON file	Error / No execution	The expected
Executing with wrong doc ID	Print empty list/Empty histogram	The expected
Executing with wrong user ID	Print empty list/Empty histogram	The expected

Table 5: Tests Conducted

In general the application does not crush or to be more precise there was none of the different scenarios tested, that crushed the application. Corrupting the JSON file will make the application ineffective, since it cannot get data from it. Several conditions could have been added to check whether a list or a histogram is empty and print a relevant message to the user, but were not crucial to the functionality of the application and were omitted. Furthermore if a user gets an empty list or an empty histogram he should consult this documentation and table 5 specifically, to identify what he might have done wrong. Empty lists and/or histograms are most probable to appear due to faulty user input.

7.Conclusions

Reflect on what you are most proud of in the application and what you'd have liked to have done differently. An optional final section of references is also encouraged.

As stated in the introduction, this coursework offered a great introduction to the Python language, getting to know some of its tools and libraries. Designing and developing an application in a new programming language, in a short time period had been a very challenging experience but gave us a chance to test our limits, improve our learning process and learn how to be effective under pressure.

Had there been more time, a GUI design would have been implemented to enhance the user interaction. Furthermore several scenarios such as wrong document/reader Ids, which result in empty lists and histograms, would have been dealt with the according conditional checks and the respective error messages, so a user can instantly know what is wrong. Also some functions could have been implemented in a more effective way and fragments of the code could have been designed in order to offer greater re usability.

To conclude with, this project offered a very intense approach to a new programming language, that required fast learning and implementation of what was learned offering overall a great educational experience.