Coursework 2

Data Analysis of a Document Tracker

Piya Gehi
H00310319

Table of Contents

Introduction

The aim of this project is to implement a Data Analysis application of a Document Tracker in Python. The application analyzes the data of the document tracker and returns results such as the countries from where the documents are read, the most popular browser, the top 10 readers as well as documents liked by other users of a particular document. The report aims to describe various aspects of the implementation.

Environment

The application developed using Python 3.9 consists of two parts - a GUI and a command line interface that has been developed using Visual Studio Code.

Organization

The sections of the report include:

- a requirements checklist to show which requirements were met,
- a discussion about the design considerations in the project in terms of the software architecture, class design as well as use of language-specific features,
- a user guide to visually represent the features of the application,
- a developer guide to display code snippets of key features of the code,
- a set of test cases to show that features work as intended,
- and ending with the conclusion which reflects on the process of building the application.

Requirements Checklist

Achieved Not Achieved

1.	The core logic of the application should be implemented in Python 3.
2.	The application should take a string as input, which uniquely specifies a document (a document UUID), and return a histogram of countries of the viewers.
3.	The application should take a string as input, which uniquely specifies a document (a document UUID), group the countries by continent and generate a histogram of the continents of the viewers.
4.	Display a histogram of all browser identifiers of the viewers.
5.	Distinguish the browser identifier by the main browser name (e.g. Mozilla), and display the result as a histogram.

6.	Determine, for each user, the total time spent reading documents and display the top 10 results.
7.	Implement an “also like” function. It takes in a document UUID and visitor UUID as input and returns the top 10 documents, using a sorting function, based on the number of readers of the same document.
8.	For the “also like” functionality, generate a graph that displays the relationship between the input document and all documents that have been found as “also like” documents
9.	Develop a simple GUI based that reads the user inputs and with buttons to process the data as required per task.
10.	The application shall provide a command-line interface to test its functionality in an automated way

Design Considerations

Overall Code Structure

The architecture used is a two layered architecture, with the front-end of the application in different classes (one for the GUI and one for the command line) and the core logic regarding the analysis of the file in another class. The GUI classes are cw2_GUI.py and cw2.py and the logic class is analysis.py. This structure ensures modularity of the code. The logic class has dedicated functions for each of the tasks.

GUI Design

cw2_GUI.py

The application is built using a Python library called Tkinter. The application has one window, with a sidebar containing buttons for the tasks. Clicking on each of the buttons opens another frame for the task, similar to opening a page on the click of a button. There are classes for each frame, with a function to display the results. The main window is a subclass to tkinter’s main window, tk.Tk, and the frames are subclasses to tk.Frame [1]. This structure is commonly used to develop Tkinter applications.

cw2.py

This module contains the command line interface to the application. The python library, argparse, is used to process the command line and have easy access to the arguments. Argparse was the library of choice over another library, Getopt, as the syntax for setting

the arguments is easier and the formatting of the help messages as well as the flags to enter is neater is handled by the library.

Logic Class Design

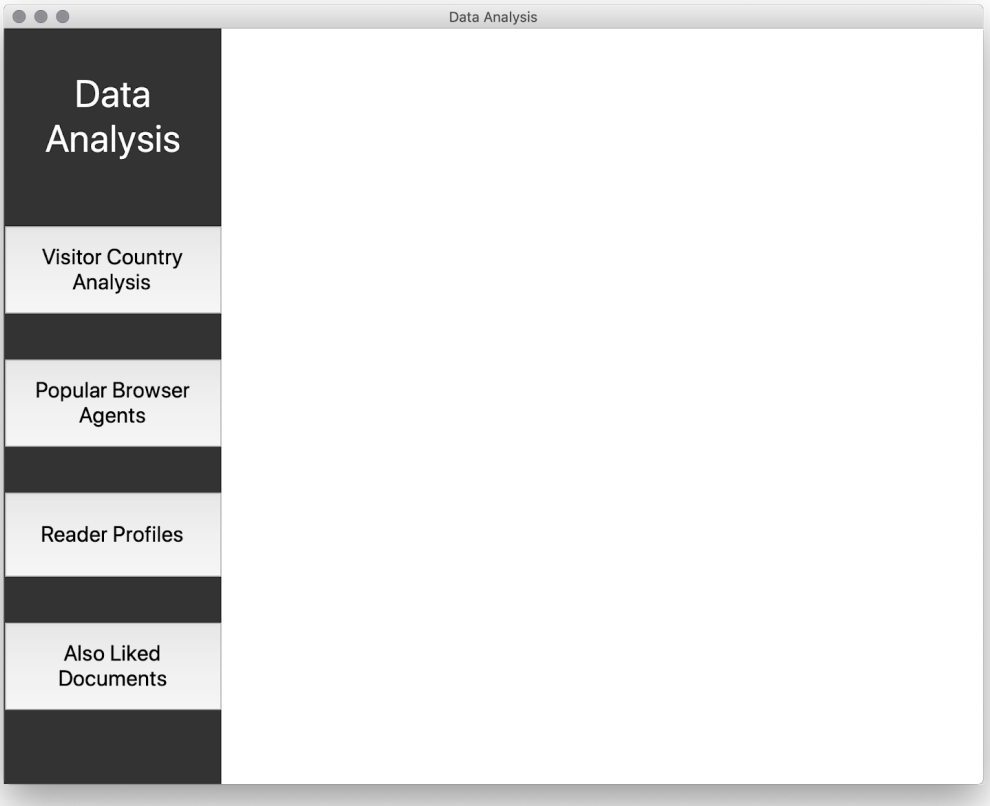
analysis.py

The module contains all the code for processing and analyzing the provided dataset. The dataset is initialized as a Pandas DataFrame in the constructor of the class, making it easier for all other methods to access. Pandas processes large datasets with ease and provides multiple functions to analyze the data, making it the library of choice over parsing with the JSON library. Each class has its own function, which takes in the required arguments required to process the dataset. Supporting functions include a function to print the graphs in tasks 2a, 2b, 3a and 3b, and a helper function for task 5 and 6 to make the code readable and reusable.

User Guide

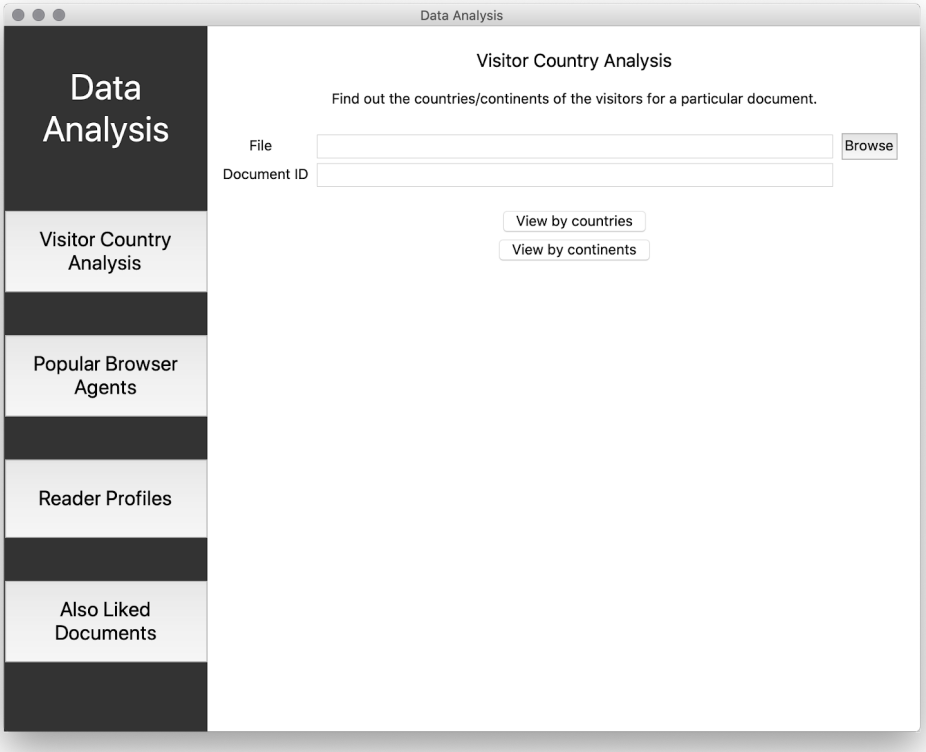
Starting the application for the first time.

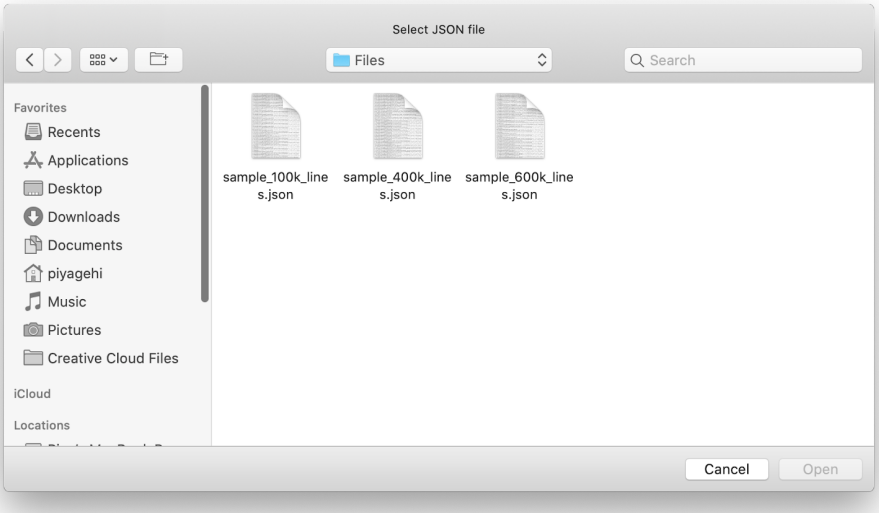
The application opens with a side bar and a blank frame. The buttons on the sidebar represent various analyses relating to the document tracker. The user can choose any one of the buttons to get started.



Visitor Country Analysis

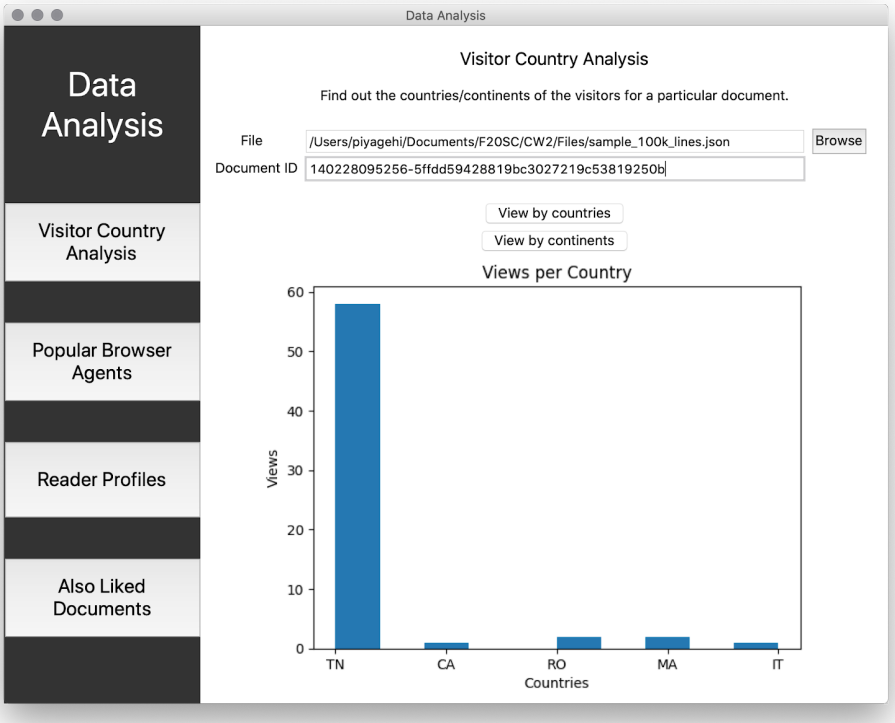
The user enters a file name or selects a file by clicking on the browse button. A document ID is also taken as input. The user has the choice to select between viewing by countries or by continents.



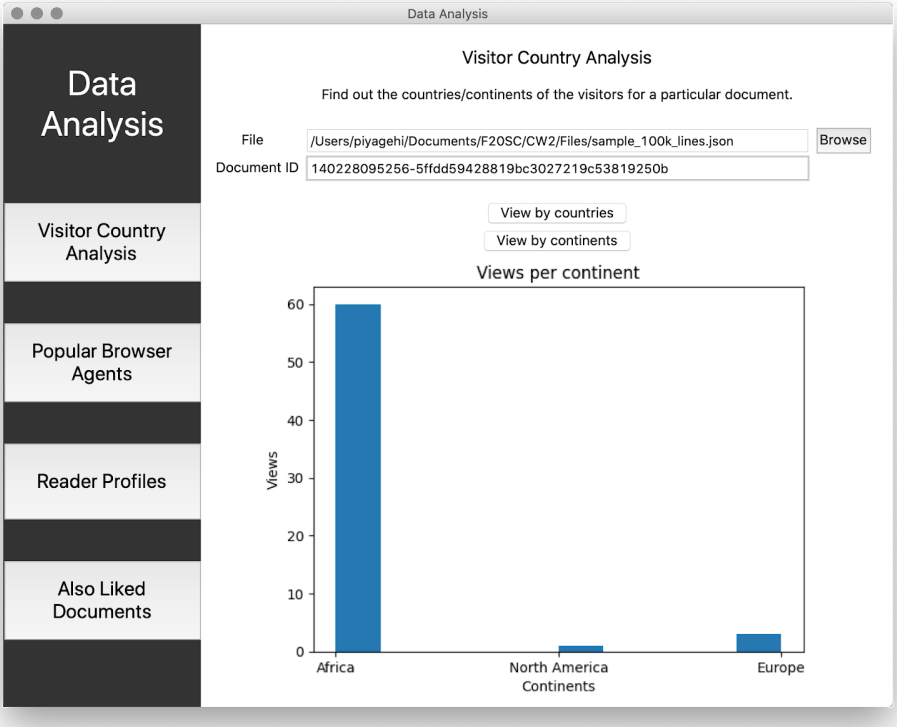


Selecting a file by clicking on the browse button

The “view by countries” option returns a histogram with a list of countries and the number of views from each country.

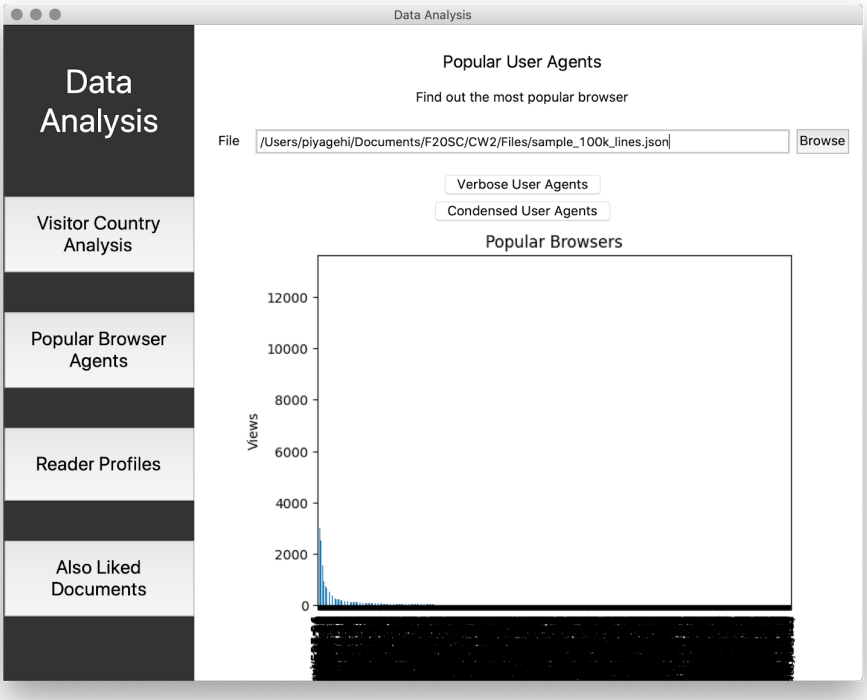


The “view by continents” option returns a histogram with the list of continents and the number of views from each continent.

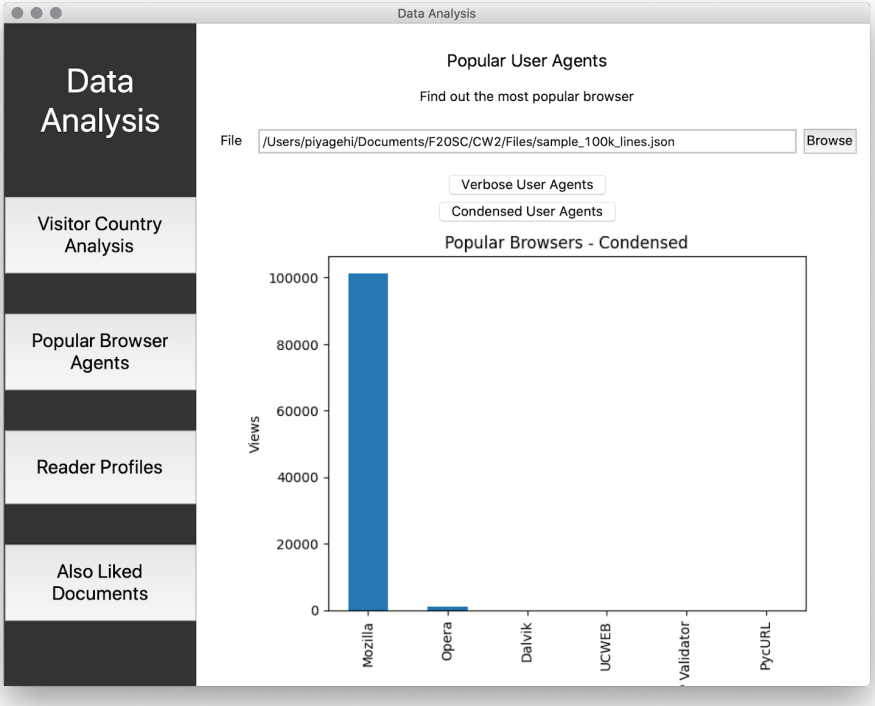


Popular User Agents

The user selects a file and enters a document ID. The user can select between detailed or verbose results and condensed results. The verbose option returns a histogram with a list of useragents (the entire string) and the number of views for each useragent.

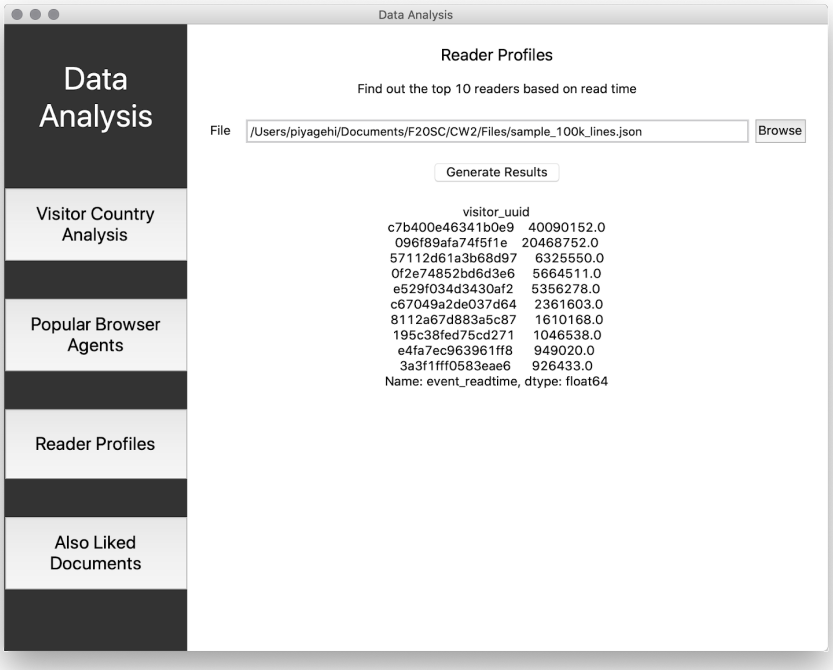


The condensed option returns a histogram with the list of shorter useragents and the number of views for each.



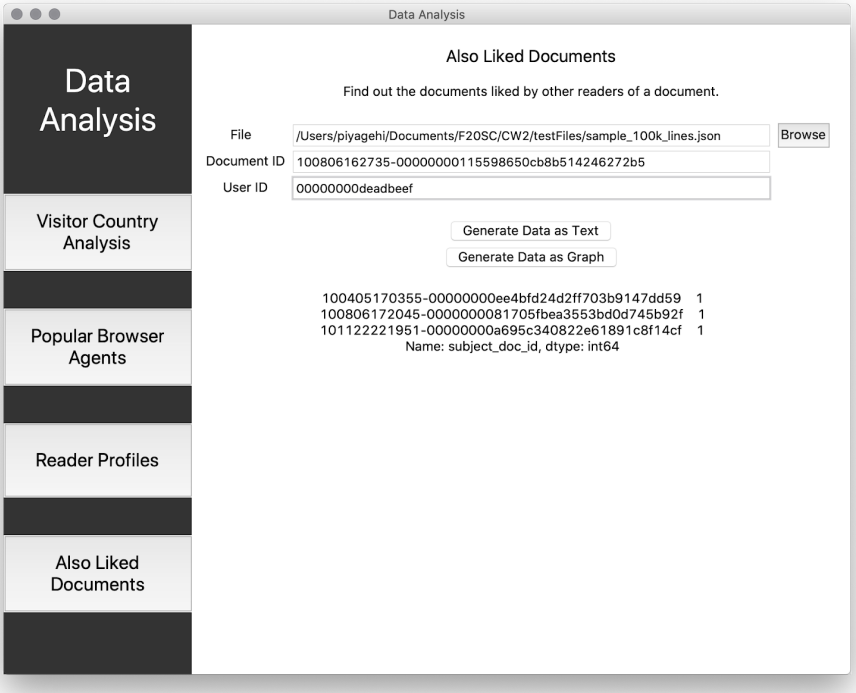
Reader Profiles

The user selects a file and clicks on Generate Results. It displays the top 10 readers in terms of read time, with the time mentioned next to the user in milliseconds.

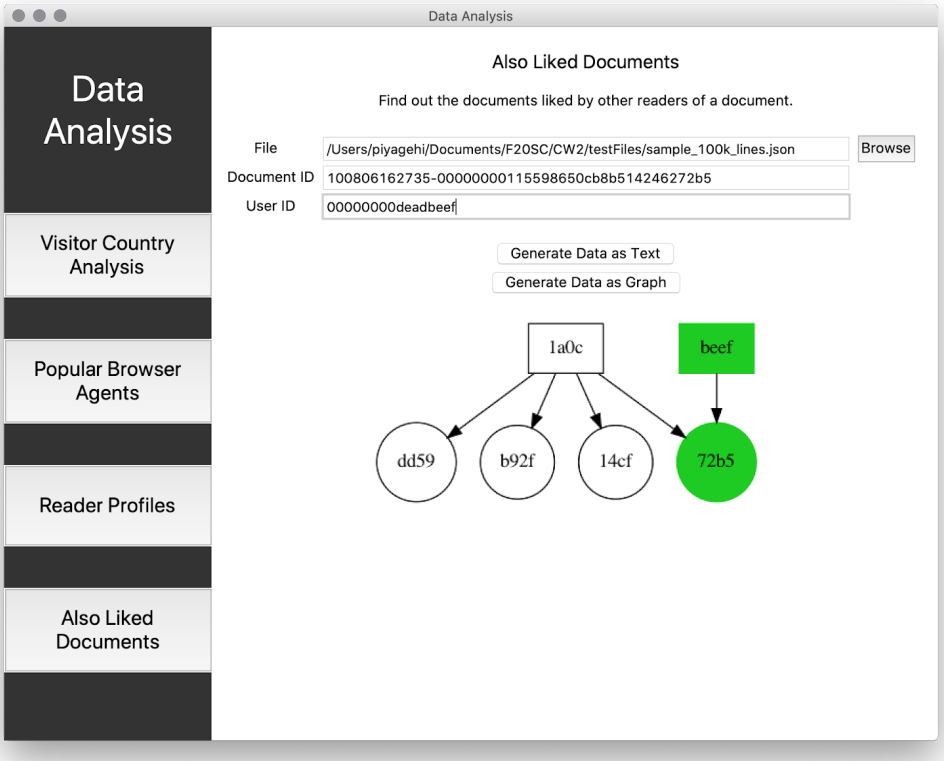


Also Liked Documents

The user selects a file, enters in a document id and optionally a user id. The user can choose to view the results as text or visually in the form of a graph. The text version returns the up to 10 documents read by users of a given document.

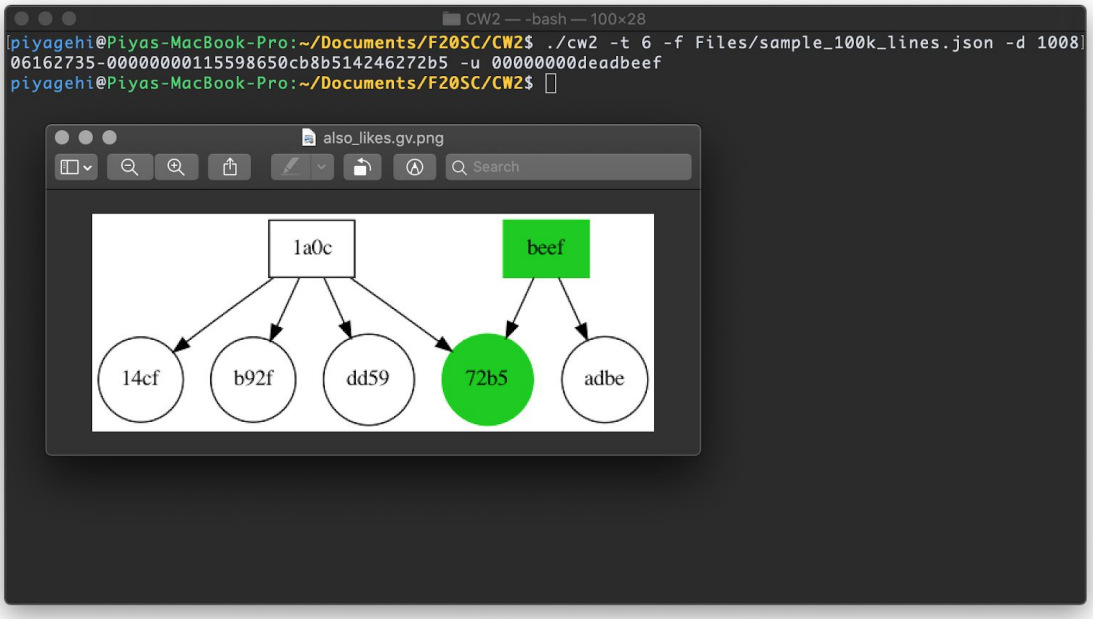


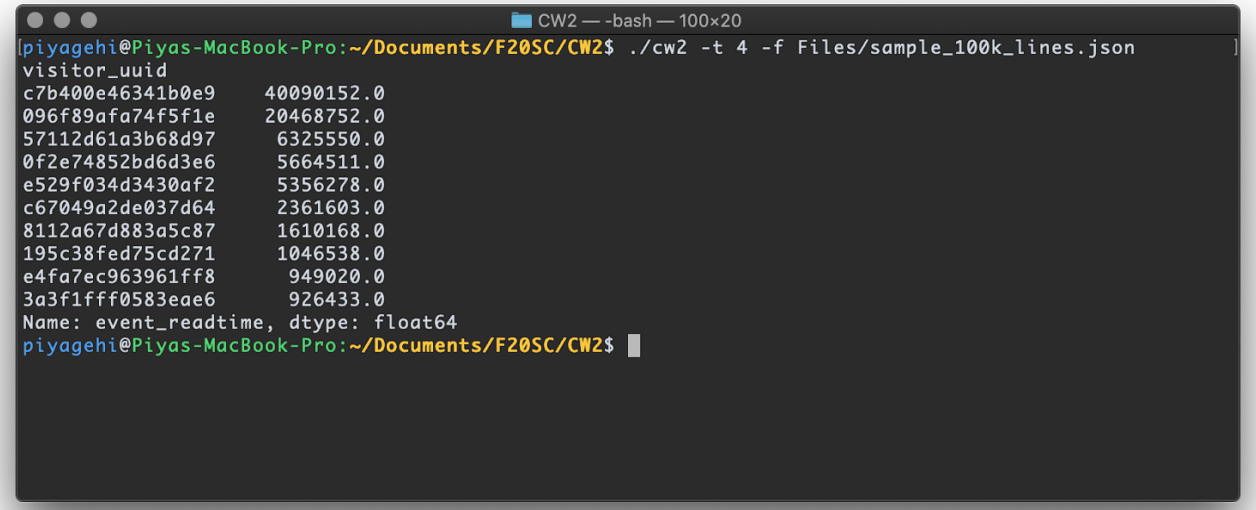
The graph version generates a graph from the textual results. The graph represents the relation of the readers with the set of documents they have read.



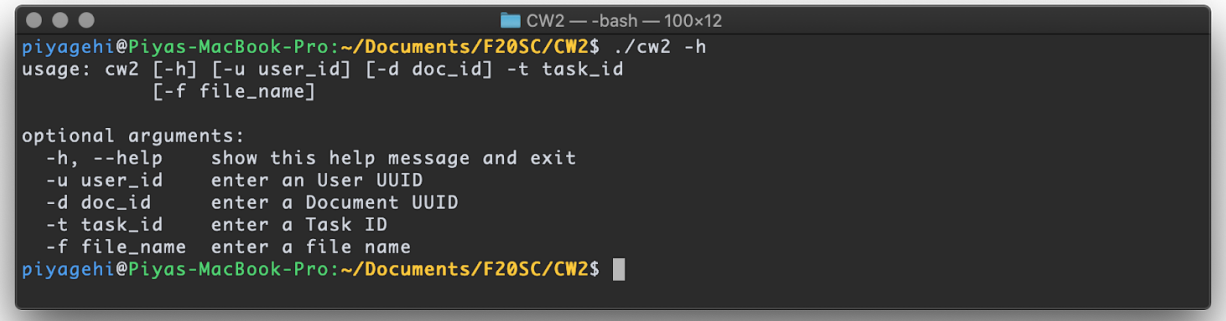
Command Line Usage

The user can access the functionality from the command line. The following command is run as an example. The user can enter the inputs after entering the following flags: -t (task-id) -f (filename) -d (document-id) -u (user-id). The resulting graphs will be opened in a new window, and any text is printed on the command line.





The user can access the list of flags and their usage by typing ./cw2.py -h.



Developer Guide

Processing the dataset

The Pandas library is used to process and store the dataset. The library contains many functions to convert file types to a structure known as a DataFrame, a two-dimensional table-like structure. One such function used here is the read_json() function to read and store the dataset. As each line in the JSON file is a separate JSON object, the lines attribute is set to True to process the information correctly. This following code snippet is from the constructor of analysis.py.

```
import pandas as pd

class Analysis():
   def __init__(self, filename):
       self.df = pd.read_json(filename, lines=True)
```

Analyzing the dataset

There are various ways to analyze the dataset. One function used throughout the analysis.py is filtering out the dataset contents based on a condition by using the square brackets or more explicitly using the .loc[] function. Specific columns can be selected by using the dot notation. Both features are shown in the following code snippet:

```
def task2a(self, inter, doc_id):
      """Returns the visitor countries for a document"""
       x = self.df.visitor_country[self.df.subject_doc_id == doc_id]
```

Another commonly used function is value_counts(), used to count the instances of unique values of a column as shown in the following code snippet. This is a handy function as it also sorts the values based on the count.

```
def task3a(self, inter):
      """Returns the visitor continents for a document"""
       x = self.df.visitor_useragent.value_counts()
```

The last function that was used in a few functions is isin(), which filters the dataframe. While .loc filters based on a condition, isin() filters based on a list of items. This function is used to implement the also likes functionality as shown in the following code snippet. Multiple columns are selected by passing it as a list, also shown in the following snippet from a function for Task 5, where readers represents the list of readers of a document.

```
self.df[self.df.visitor_uuid.isin(readers)]..where(self.df.event_type == 'read')
```

Printing matplotlib graphs

A separate function called print_graph is created in analysis.py to display graphs made using the matplotlib library. It takes in details such as the graph title, labels for the axes, and the type of the graph. Based on the type of graph, the functions are called.

```
import matplotlib.pyplot as plt

def print_graph(self, x, title, xlabel, ylabel, ptype, filename, inter):
       """Plot and display matplotlib graphs"""
       plt.clf()
       plt.ylabel(ylabel)
       plt.xlabel(xlabel)
       plt.title(title)
       if ptype == 'bar':
           x.plot.bar() # value_counts() function has a plot function which is
used here
       elif ptype == 'hist':
           plt.hist(x)
```

The inter variable is to change the output depending on whether the graph is called by the command line or by the GUI. When the function is called from the GUI, it is saved as an image which is then displayed in a label in the GUI. Otherwise, the .show() function is used to display the graph when called from the command line.

```
       if inter == 0:
           plt.savefig(filename, bbox_inches='tight')
       else:
           plt.show()
```

Also likes functionality and graph

The also likes functionality is implemented using 3 functions - also_readers_docs(), task5d() and task6().

also_readers_docs() is a helper function that combines tasks 5a, 5b and 5c. It first generates a list of unique readers of a given document, then generates a list of documents read by those readers. The function also removes any duplicate and null values that may be generated.

```
def also_readers_docs(self, doc_id):
       """A helper function to return the list of document"""

       #list of readers of a document
       readers = self.df.visitor_uuid[(self.df.subject_doc_id == doc_id) &
(self.df.event_type == 'read')].unique()
```

```
      #list of users and the documents they've read
       return self.df[self.df.visitor_uuid.isin(readers)][['subject_doc_id',
'visitor_uuid']].where(self.df.event_type == 'read').drop_duplicates().dropna()
```

task5d() displays the results from also_readers_docs. Before displaying the values, however, the current doc_id is filtered from the list. Then the top 10 documents are displayed using the value_counts() function, which also acts as a sorting function in this case. As the task stores a copy of a part of the DataFrame, the original DataFrame is not altered in the process.

```
def task5d(self, inter, doc_id, visitor_uuid=None):
       """Prints top 10 liked documents by readers of a given document"""

       #this task displays documents apart from the given doc_id
       doc_list =
self.also_readers_docs(doc_id).subject_doc_id[((self.df.subject_doc_id !=
doc_id) & (self.df.visitor_uuid != visitor_uuid))]

       #documents are printed based on the number of readers of the document
       return doc_list.value_counts().head(10)
```

task6() displays the graph based on the results of task 5d(). However, as value_counts only returns the count, there is no way to retrieve the list of who read the documents in the list from task5d(). This is why the also_readers_docs() function is called here, which provides the list of visitors as well as a mapping of doc_id to visitor_id, which is beneficial in creating the graph. The graph is a directed graph from the graphviz library, and the resulting graph is saved as a png as specified in the format.

```
def task6(self, inter, doc_id, visitor_uuid=None):
       """Prints the results from task5d() in a graph format"""

       dot = Digraph(comment='Readers of this document also like.',
format='png')

       #retrieve the list of top 10 documents
       docs = self.task5d(inter, doc_id, visitor_uuid).index.tolist()

       #add the document id to show the input document in the graph
       docs.append(doc_id)

       #stores the list of the doc_id and the visitors who read the document
       doc_vis = self.also_readers_docs(doc_id)
       doc_vis = doc_vis[doc_vis.subject_doc_id.isin(docs)]

       #retrieves a unique list of visitors
       visitors = doc_vis.visitor_uuid.unique()
       if visitor_uuid:
           numpy.append(visitors, visitor_uuid)
```

The second part of the function is the printing of the graph. For readability, both user and document nodes only display the last 4 characters of the ID.

User nodes, denoted by a box shape, are generated by iterating through the visitors list. The visitor_uuid, if entered by the user, is highlighted in a different color.

```
      #iterate visitors list and generate the visitor nodes, including the
      visitor_uuid if entered
       for r in visitors:
           if (visitor_uuid is not None) and (visitor_uuid == r):
               dot.node(r, r[-4:], shape='box', style='filled', color='green3')
           dot.node(r, r[-4:], shape='box')
```

Similarly the document nodes, denoted by the circle shape, are generated by iterating through the documents list. The doc_id is highlighted in a different color.

```
      #iterate documents list and generate the document nodes
       for d in docs:
           if d == doc_id:
               dot.node(d, d[-4:], shape='circle', style='filled',
color='green3')
           dot.node(d, d[-4:], shape='circle')
```

The edges between the user and visitor id's are then created by iterating through the doc_vis list:

```
      #draws an edge from each visitor to each document in the doc_vis list
       for x in doc_vis.values:
           dot.edge(x[1], x[0])
```

The graph is then rendered and displayed in a separate window when called by the command line. If it's called from the GUI, then the resulting image is displayed in a label.

Testing

Six unit tests were run testing all the core functionality of the system with the sample_100k_lines.json file as the dataset. All the tests were passed as seen in the following screenshot. One particular test took a while to run (task3a), as the user agent strings are very long and generating a graph takes a while.

```
piyagehi@Piyas-MacBook-Pro:~/Documents/F20SC/CW2$ /Library/Frameworks/Python.framework/Versions/3.9/bin/pyth
on3 /Users/piyagehi/Documents/F20SC/CW2/test.py
.......
----------------------------------------------------------------------
Ran 7 tests in 85.672s

OK
```

What did I learn from CW1?

The coursework 1 feedback suggested improvements in the code quality and the report. The report suggestions were specifically for the design considerations and the developer guide, which have been taken into consideration while making this report. The developer guide explains some of the features used and why they were used, and the design considerations also explains the choices and why they were chosen.
The feedback for the code quality emphasized on maintaining modularity and reusability of code, which was achieved by maintaining separate classes for the core logic, GUI and command line interface. Apart from the feedback, I did realize the importance of time management and planning out certain aspects (GUI Design for example) before implementing them.

Conclusion

The application aims to analyze data of a document tracker. The application covers all the functionality specified using the specified libraries. I'm most proud of the GUI interface, as it took a while to get the hang of it. The functionality has been tested multiple times, through unit testing as well as during debugging. Developing applications in scripting and systems languages has its own strong and weak points. Scripting languages like Python are known to have a strong set of libraries that a user can work with, however building an application with a system language like C# felt more convenient and intuitive. Functions in Python performed a lot of functionality with a few lines of code, whereas in systems languages more lines were required. Both languages are suited for different purposes.

References

[1] https://stackoverflow.com/questions/7546050/switch-between-two-frames-in-tkinter

Appendix

There are a few errors in the screencast that were noticed after the recording was done. The following errors were fixed later in the application and the report:
1. The text and graph versions for also-likes were slightly different from the test results provided to us, the updated version of the results is part of the user guide screenshots.
2. The button on the sidebar reads "Popular Browser Agents", while the heading on the page and my voiceover mentioned "Popular User Agents". Hence the button text has been changed to "Popular User Agents" in the final GUI.