

U.S. State Tourism

Final Report

Abhinav Verelly, Ashutosh Bhattarai, Shane Grishaw, David Gruhn

CS 4624: Multimedia, Hypertext, and Information Access

Virginia Tech
Blacksburg, VA 24061

May 9, 2021

Instructor: Dr. Edward A. Fox

Client: Dr. Florian Zach

Table of Contents

Table of Figures	
Table of Tables	
Abstract - Executive Summary	1
Introduction	2
Teams Roles	3
Requirements	3
Data Extraction	3
Visualizations	4
Design	4
Extractor	5
Meta Tags CSV	5
External / Internal links CSV	5
Images CSV	6
Colors / Background Images CSV	6
Raw Text CSV	6
Custom Data Parser	6
Multi-File Processing	7
Implementation	7
Coding Environment	7
Language + Tools	7
Loading All Data + Config File	7
Extracting External / Internal Links	8
Extracting Images + Trackers	9
Extracting Raw Text	9
Extracting Colors	10
CSV Output - makeCSV(filename, columns_name, tabellInfo)	10
Log File	11
Visualization - Stacked Bar Chart	11
Visualization - Extracting Color from CSV Files	11
Visualization - Filtering Out / Validating Colors	11
Visualization - Getting the Frequency of the Colors	12
Visualization - Creating the Stacked Bar Chart	12
Visualization - Outputting JSON File	13
Assessment of Implementation	13
Jupyter Notebook	13

Data Extraction Process	14
Time Complexity	15
User's Manual	15
Developer's Manual	20
Jupyter Installation	20
Library Installation	20
Loading files, Extraction, and CSV output	22
Handling Visualization - Extracting Color CSV Files and Storing	26
Methodology	31
Goals of our Users	31
Subtasks of our goals	32
Implementation-based services	32
Workflows	35
Lessons Learned	36
Timeline / Schedule	36
Problems	37
Solutions	38
Future Work	38
Acknowledgements	40
References	41

Table of Figures

Figure 1: ER Diagram depicting the system created	5
Figure 2: Example of a Completed JSON File	17
Figure 3: Colorado's Completed Stacked Bar Chart	17
Figure 4: California's Completed Stacked Bar Chart	18
Figure 5: Virginia's Completed Stacked Bar Chart	18
Figure 6: Example of California's Legend for the Stacked Bar Chart	19
Figure 7: Installing the required libraries	21
Figure 8: Running the cells in Jupyter Notebook	21
Figure 9: Our headerAndTime() method	22
Figure 10: loadFile() and reading in all Parquet file	23
Figure 11: directoryToChange variable that users need to change	23
Figure 12: Outputs each CSV to unique folder	23
Figure 13: Output of each Parquet file into unique directories	24
Figure 14: example of contents in output folder	24
Figure 15: makeCSV method and its usage	25
Figure 16: Sample output of log.txt	26
Figure 17: Load all Output Directories	26
Figure 18: Grabbing data from color CSV Files	27
Figure 19: Checking the colors extracted and validating them	28
Figure 20: Validating a color into Hex-Color format	28
Figure 21: Gets the frequency of the colors in a year	29
Figure 22: Gets the percentage of colors out of the total colors in a year.	29
Figure 23: Outputs a JSON File of the frequency type nested dictionary	30
Figure 24: Outputs a stacked bar chart and legend corresponding to the graph.	30
Figure 25: Gets the state name from the config file	31
Figure 26: Workflow Image	36

Table of Tables

Table 1: Group members names and respective roles	4
Table 2: Total Time For Extraction	23
Table 3: Implementation Service Table	32
Table 4: Project Schedule	35

I. Abstract - Executive Summary

Each state in the United States has its own state-run website, which is used as a means to attract new tourists to that location. Each of these sites is typically used to highlight any big attractions in that state. Any travel tips, facts regarding that location, blog posts, ratings from other individuals that have traveled there, or any other useful information that may attract potential tourists are also included. These websites are maintained and funded directly by occupancy taxes. Occupancy taxes are a form of state tax that an individual pays whenever one stays in a hotel or visits any attractions in that state. As such, the main goal of these websites is to attract new tourists to their location. These websites are maintained and paid for by past tourists who have visited that state.

Funding for future state tourism is determined by how many previous tourists have visited the state and paid the occupancy tax. Researchers need to be able to determine which elements of the website are most beneficial in attracting tourists. This can be determined by examining past tourism websites and looking for any patterns that would determine what worked well and what didn't. These patterns can then be used to determine what was successful and use that information to make better-informed decisions.

Our client, Dr. Florian Zach of the Howard Feiertag Department of Hospitality and Tourism Management, plans to use the historical analysis done by our team, to further help his research on trends in state tourism websites content. Different iterations of each state tourism website are stored as snapshots on the Internet Archive and can be accessed to see changes that took place in that website. Our team was given Parquet files of these snapshots for the states of California, Colorado, and Virginia dating back to 1998. The goal of the project was to assist Dr. Zach by using these Parquet files to perform data extraction and visualization on tourism patterns. This can then be expanded to other states' tourism websites in the future.

We used a combination of Python's Pandas library, Jupyter Notebook, and BeautifulSoup to examine and extract relevant pieces of data from the given Parquet files. This data was extracted into various different categories, each with its own designated folder. These categories were raw text, images, background colors and background images, internal and external links, and meta tags. With this data sorted into the appropriate folders, we are then able to determine specific patterns such as what colored background was used the most. With our data extraction portion of this project completed along with the visualization, we hope to pass this on to future teams so that they are able to expand on our current project for the rest of the states.

II. Introduction

Our goal for this project is to be able to extract data from past and present iterations of the California, Colorado, and Virginia state tourism websites, www.visitcalifornia.com, www.colorado.com, and www.virginia.org, respectively. Researchers would then analyze the data to make valuable decisions regarding future iterations of the state website. This goal can be broken down into two categories, one for each of our clients, Dr. Florian Zach and Dr. Edward A. Fox, the professor of the CS 4624 (Multimedia, Hypertext, and Information Access) capstone course. Dr. Florian Zach asked us to produce a system that is able to extract as much information as possible from the previous iterations of the state website. The information extracted includes raw text, images, background colors with background images, internal and external links and meta tags. We would then store the information in a location where there are patterns in the data to make more informed decisions for future iterations of tourism websites. We created a visualization model that can be presented to the class. For this goal, we chose to focus on the colors that are present within each iteration of the website to determine the frequency of each color being used.

For both goals, it was crucial that we were able to extract the data first. We were given Parquet files of each iteration of Colorado, Virginia, and California websites to extract data from. We decided to start working with the Colorado Parquet files first. These Parquet files contained snapshots of each version of the Colorado website from the 1990s to 2019. After becoming familiar with the types of information that were present in these Parquet files, we determined and planned the extraction process with Dr. Florian Zach requested. We created a parser using Python's Pandas library, Jupyter Notebook, and BeautifulSoup which would sift through all of the Parquet files and pull out any relevant information we desired [2][3][4][5].

We organized each of our data into their own CSV files from each Parquet file. We analyzed and placed them within their own respective folders (e.g., all raw text for each Parquet file went in the raw text folder). This organization would help Dr. Florian Zach and future teams to pick up our work or gather the right information. In the visualization aspect of our project, we grabbed all the color CSV files and then plotted a stacked bar chart. The chart shows how much of each color was used within that year from the tourism websites. Our hope is that the work we have completed, can allow future teams to expand and further our work through extracting data and visualization.

A. Teams Roles

Our team was divided into three different roles, the Project Lead, Data Extractor, and Data Visualizer (Table 1). The Project Lead worked as a liaison between Dr. Florian Zach, Dr. Edward A. Fox and our team. The Project Lead also relayed any new or pertinent information to the team in between scheduled meetings. The Data Extractor role was responsible for developing the code that parses through the Parquet files and creates designated CSV files for the desired data and places them into their own designated folder. The Data Visualizer roles were responsible for taking the extracted data and creating stacked bar charts with that data to better help visualize the results. While each individual on the team had their own specific role, all team members were present and helped out with each portion of the project and worked together to produce any reports or presentations.

Team Member	Role
Abhinav Verelly	Project Lead, Data Visualizer and Project Manager
Ashutosh Bhattarai	Data Extractor (Meta Tags, Raw Text),
David Gruhn	Data Extractor (Colors, Images, Links), Meeting Scribe
Shane Grishaw	Data Visualizer, Data Processor

Table 1: Group members names and respective roles

III. Requirements

We will utilize a dump of website Parquet files, extracted from the previous team's WARC files of tourism websites. These snapshots will be in the form of file directories of these Parquet files. We will then need to parse this data from the files using Python, BeautifulSoup, and Pandas [2][3][5]. There will be a data extraction and visualization part for this project.

A. Data Extraction

These Parquet files, mentioned above, are given to us by the previous year's team. The refactoring of these Parquet files is necessary, as it is difficult to read them. We were then tasked to extract relevant information, pointed out by Dr. Florian Zach. Main

extraction elements included colors, image tags, meta tags, external/internal links, and raw text of the files. We extracted specific items from these elements, like the word count and the timestamp. The data was extracted using Python and its external libraries [7]. Python's Pandas helped read these Parquet files for the extraction. BeautifulSoup was used to extract specific elements from these Parquet files [2][3][5].

B. Visualizations

The main prerequisite for running the visualization program is to have previously run the data extractor on the Parquet files [2]. This is needed so we can accurately pull the data needed to populate the chart. We used a stacked bar chart to view our data extraction and structured the chart by years for all tourism websites. The previous group structured their graphical representations to include seasonal visualization. This visualization showed the changes occurring by months. We did not follow this approach, as there would be a lot of graphs for one state. We would have to split each year to have a separate stacked bar chart. The data from these tourism websites range from the 1990s to 2019. Therefore, having to make a large amount of these graphs for one state may be cumbersome. However, we were still able to see patterns of how colors were used within each state by their specific year.

IV. Design

This section will focus on how the project was designed. Figure 1 shows an extended ER Diagram (Entity-Relationship Diagram), that depicts how data is organized and how the whole system works.

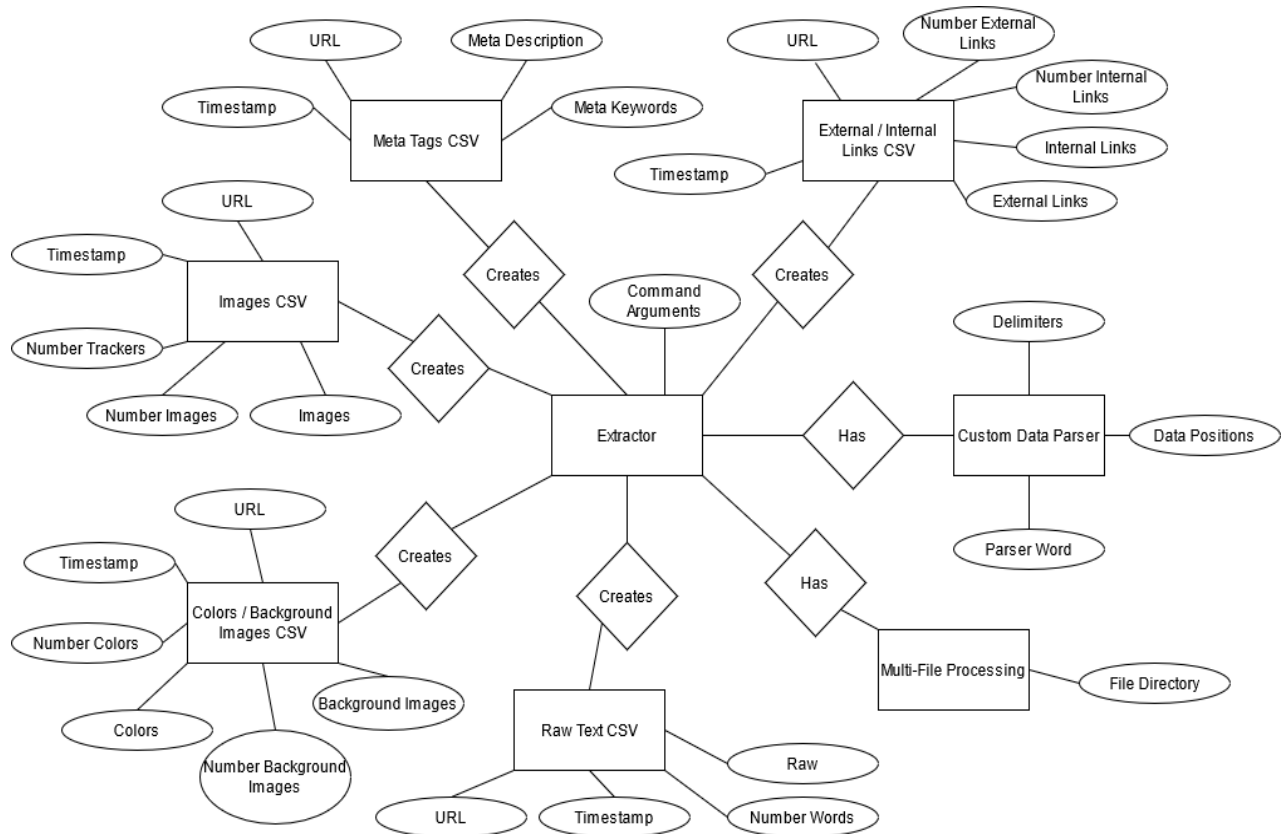


Figure 1: Extended ER Diagram depicting the data organization and system created

A. Extractor

Our main entity that branches out throughout our design is the Extractor, located in the center of Figure 1. This represents the main part of the code that takes in the parameters and necessary files to extract the data. It has a singular attribute “Command Arguments”, which represents the website's base name. For example, `www.google.com` would represent the base website. Any links containing a base website will be filtered accordingly in the respective entities.

B. Meta Tags CSV

The Meta Tags CSV entity is connected to the Extractor entity with a “Creates” relationship. This will be created by the Extractor with the following attributes. “Timestamp” and “URL” serve as unique identifiers for the data that is extracted. “Meta Description” and “Meta Keywords” serve as the desired data from the Parquet file to be examined.

C. External / Internal links CSV

The External / Internal Links CSV makes use of the “Command Arguments”. They are passed into the extractor to determine the difference between external and internal links. It also includes the same unique identifiers that Meta Tags CSV utilizes. It has two lists with two counters. Each list has a respective counter. For example, “Number External Links” denotes the number of links in the list provided by “External Links.” Likewise can be said for the internal link attributes “Number Internal Links” and “Internal Links.”

D. Images CSV

The Images CSV shares similarities with the previous two entities discussed above. It contains the unique identifiers “URL” and “Timestamp.” It also contains a list of images represented by the “Images” attributes with a respective number of images in that list, designated by the “Number Images” attribute. Lastly, it contains a unique attribute denoted as “Number Trackers” that counts the number of pixel images found while extracting the images.

E. Colors / Background Images CSV

The Colors / Background Images CSV is very similar to its counterpart External / Internal Links CSV entity. It contains the unique identifiers “URL” and “Timestamp.” In addition, it contains two lists as attributes “Number Colors” and “Background Images,” which respectively have a counter denoted by the attributes “Number Colors” and “Number Background Images.”

F. Raw Text CSV

The Raw Text CSV Entity contains the same unique identifiers “URL” and “Timestamp.” The “Raw” attribute contains all the extracted raw text, which is sorted as a list of words. The extracted raw text contains a counter that denotes how many words are contained in the list of raw text. The length of the list is denoted by “Number Words”.

G. Custom Data Parser

Moving towards entities that have a relationship of “Has” with the Extractor, we find the Custom Data Parser entity. Please take a look at “Assessment of Implementation” to understand more about this entity. It is not abstracted from the bulk of the code, but easily could be. One attribute it has is “Delimiters” to determine where to stop extracting. It also contains “Parser Word,” which is the wanted attribute to be searched

for in the Parquet file. Lastly, we have “Data Positions,” which will be updated to show where the found “Parser Word” is in the data.

H. Multi-File Processing

Similar to that of the Custom Data Parser, the “Multi-File Processing” entity has a “Has” relationship with the Extractor entity. It has one attribute that is “File Directory” which refers to the path where the data will be located and saved after final executions. Similar to that of Custom Data Parser this is not abstracted from the code, but easily can be.

V. Implementation

This section will focus on the implementation and design choices we made to extract the data that was wanted by our client, Dr. Florian Zach.

A. Coding Environment

We were given the past group’s codebase, where they had utilized Jupyter Notebook as their main coding environment. Although none of us were familiar with Jupyter Notebook, for consistency purposes we decided to utilize it. Our client, Dr. Florian Zach also had familiarity with Jupyter Notebook which would make passing off our result easier. Jupyter Notebook is an IDE where there can be different cells for each line of code/or a whole code block. Each cell can output different things depending on what each cell contains. This allows us to run each cell that contains different outputs (such as file names or debugging statements). In addition, we can easily make new changes and see our updated results in real-time [4].

B. Language + Tools

Continuing with the given codebase from last year’s team, they utilized Python and various libraries of Python which we decided to employ too. Python is a powerful language that would allow us to output data in our choice of format, parse through HTML, and allow us to perform data analysis on extracted data [7][10].

- Anaconda: Allows Windows users access to the code base [6]
- BeautifulSoup: Will allow us to parse HTML payloads [5]
- Pandas: Will allow us to read in Parquet files + output data into CSVs [3]

C. Loading All Data + Config File

The original codebase's design required us to manually input the name of each Parquet file. We decided to change the design to be more abstract and dynamic by requiring a file path to the location of all the Parquet files. The user enters their file path, and the program opens all Parquet files and starts reading them. You can change the host website name in the config file. Please refer to the configuration file to understand proper formatting. A brief example is to input the main host website name as `www.google.com` inside the `config.txt`. Then our code will check each Parquet file's links contained therein. If all links have the same hostname as written in the `config.txt`, these links are considered internal links. Any other links with different host names will be considered external links, e.g., `www.youtube.com`. Likewise, the file path of where the raw data is will need to be entered manually into the code at the top. All this goes into our `headerAndTime(df, webName)` method. We take in a variable called `df`, which stores the Parquet file that we open using a Pandas object. Then we call the function `.read_Parquet()` on the Pandas object to read in the Parquet file [2][3].

D. Extracting External / Internal Links

Regarding extraction of all external and internal links, we utilize BeautifulSoup to parse through the HTML in the Parquet file. The information we are extracting for each row of the CSV file will be stored in a list. When naming our CSV files, we grab the timestamps from each Parquet file, and we find the oldest date and the youngest date. For example, the current Parquet file we are reading grabs 19980423 as the oldest timestamp, and 2018081 as the youngest timestamp. We then would use this to name our CSV file in the format of `oldest_youngest_website_datapoint.csv`. Using our above example, we would name our CSV file as `19980423-20180814_www.colorado.com_ExtIntLink.csv`. We will also have a dictionary that will store specific information corresponding to each header in our CSV file for our link extraction. We have the following headers for links: `'Timestamp'`, `'URL'`, `'Number_ExternalLinks'`, `'ExternalLinks'`, `'Number_InternalLinks'`, and `'InternalLinks'`. In our dictionary we store each information item into its specified header category (e.g., `linkDict['Number_Internal_Links'] = numInternal`). Using `find_all()` method, we look for all body tags within the Parquet file as that is where all links will be contained. Furthermore, all links in HTML are entered in the format of `a="link"`; therefore we once again find all instances of `'a'` within all the body tags. We assume that all links will start with `http://` and we start grabbing all links. Once we have grabbed everything we want, we append a dictionary to the list we made (e.g., `linksInfo.append(linkDict)`). Our dynamic method for CSVs handles outputting a unique CSV for each data point that we are extracting. This method allows us to create a CSV for external and internal links [2][5][10].

E. Extracting Images + Trackers

Regarding extraction of all images, we also utilize BeautifulSoup to parse through the HTML in the Parquet file. For naming of our CSV files, we grab the timestamps from each Parquet file, and we find the oldest date and the youngest date. For example, the current Parquet file we are reading grabs 19980423 as the oldest timestamp, and 2018081 as the youngest timestamp. We then would use this to name our CSV file in the format of oldest_youngest_website_datapoint.csv. Using our above example, we would name our CSV file as 19980423-20180814_www.colorado.com_Images.csv. As before, the information we are extracting for each row of the CSV file will be stored in a list. We will also have a dictionary that will store specific information corresponding to each header in our CSV file for our image extraction. We have the following headers for images: *'Timestamp', 'URL', 'Number_Trackers', 'Number_Images', 'Images'*. In our dictionary we store each information into its specified header category (e.g., *imageDict['Number_Images'] = numImage*). Once again we use BeautifulSoup's *find_all()* method to find all images in HTML. One thing of note we discovered as we extracted images were these entities we dubbed as trackers. These trackers are embedded into images as pixels, and we later learned that they were trackers (hence why we called them trackers) for advertisements. We determine if an image is a tracker based on two slashes (//) whereas an image itself is one slash (/). Once we have grabbed everything we want, we append a dictionary to our list (e.g., *imageInfo.append(imageDict)*). Our dynamic method for CSVs handles outputting a unique CSV for each data point that we are extracting. This method allows us to create a CSV for images and trackers [2][5][10].

F. Extracting Raw Text

Regarding extraction of all raw text, we also utilize BeautifulSoup to parse through the HTML in the Parquet file. For naming of our CSV files, we grab the timestamps from each Parquet file, and we find the oldest date and the youngest date. For example, the current Parquet file we are reading grabs 19980423 as the oldest timestamp, and 2018081 as the youngest timestamp. We then would use this to name our CSV file in the format of oldest_youngest_website_datapoint.csv. Using our above example, we would name our CSV file as 19980423-20180814_www.colorado.com_Raw.csv. As before, the information we are extracting for each row of the CSV file will be stored in a list. We also have a dictionary that will store specific information corresponding to each header in our CSV file for our raw text extraction. We have the following headers for images: *'Timestamp', 'URL', 'Number_Words', 'Raw'*. In our dictionary we store each information into its specified header category (e.g., *rawDict['Number_Words'] = wordCount*). We use the body of our BeautifulSoup

object to find all raw text by using the `get_text()` method, as we want to ignore the meta tags and payload from the Parquet file. In addition, we use regex to filter out extraneous and special characters (*re.sub('(?! \d)[,; > < » > < ... / + | & © /] (?! \d)', '', text)*). Once we have grabbed everything we want, we append a dictionary to our list (e.g., *rawInfo.append(rawDict)*). Our dynamic method for CSVs handles outputting a unique CSV for each data point that we are extracting. This method allows us to create a CSV for raw text [2][5][10].

G. Extracting Colors

Regarding extraction of colors, we had to make our parser as there wasn't a built-in BeautifulSoup method to grab all colors the way we intended for this project. For the naming of our CSV files, we grab the timestamps from each Parquet file, and we find the oldest date and the youngest date. For example, the current Parquet file we are reading grabs 19980423 as the oldest timestamp, and 2018081 as the youngest timestamp. We then would use this to name our CSV file in the format of `oldest_youngest_website_datapoint.csv`. Using our above example, we would name our CSV file as `19980423-20180814_www.colorado.com_Colors.csv`. As before, the information we are extracting for each row of the CSV file will be stored in a list. We also have a dictionary that will store specific information corresponding to each header in our CSV file for our raw text extraction. We have the following headers for images: 'Timestamp', 'URL', 'Number_Colors', "Colors". In our dictionary we store each information into its specified header category (e.g., *colorDict['Number_Colors'] = colorCount*). Our parser goes through all of the HTML code and looks for any instances of the word “*bgcolor*” and “*background-color*”. We then read what is immediately after these words and store that information as the color. Once we have grabbed everything we want, we append a dictionary to our list (e.g., *colorInfo.append(colorDict)*). Our dynamic method for CSVs handles outputting a unique CSV for each data point that we are extracting. This method allows us to create a CSV for colors [2][5][10].

H. CSV Output - makeCSV(filename, columns_name, tableInfo)

Regarding outputting all files, we made a method in order to make this aspect modular. We passed in the filename that we wanted for each CSV file output (filename). Then we passed in all the headers that we wanted in the CSV file (columns_name). Lastly, we passed the information we are extracting for each row of the CSV file which we store in a list (tableInfo).

I. Log File

Regarding our log file, we write what Parquet file we have finished processing so far (Figure 11). This allowed us to debug in case our code broke; we would know which Parquet file caused the issue. We simply used the built in methods in Python to read and create a file as we were extracting each Parquet file [2][7].

J. Visualization - Stacked Bar Chart

After speaking with the client, Dr. Florian Zach, we decided to display our data using a stacked bar chart. It has weights within a bar that correspond to the amount of color shown for a specific bar. We decided to make the chart by using the years from the Parquet File's website data [2]. We used years instead of months, because the graphs would be a lot larger and the color frequency data would be cluttered. It was also more convenient to have a chart for each state's color visualization, rather than multiple charts for a specific year. More visuals on how the stacked bar chart is generated can be shown in Figure 21.

K. Visualization - Extracting Color from CSV Files

We have a file reader that goes through and finds all the "color.csv" files from each output directory. The method "loadFile()" (Figure 17) checks for the Parquet file and will store these CSV files in a list to open all files. We use a for loop to go through the list of all Parquet files. We then check for "colors.csv" files, and make a dictionary with the key being the year of the websites and the value being the color for that associated year. The method "calculate()" does this exactly. This method will create a dictionary that will hold the year and color information from the methods above [2].

L. Visualization - Filtering Out / Validating Colors

In the "colorConvert_Frequency()" method, we go through the value of the dictionary above and check if the color is in the right Hex-Color format, using Matplotlib's API. We use Matplotlib for colors that appear to be in word form rather than traditional Hex-Color format. We use the method "matplotlib.colors.cnames["color"]" to get the Hex-Color format. If it's not in color format, we validate that it is in the right Hex-Color format. Some CSS style sheets display color tags in unique ways. One case we found with CSS sheets, was the instance of "TRANSPARENT". This is typically the color's opacity and doesn't necessarily have a color designation. Dr. Florian Zach thought it would be a great idea to include this within the color visualization. This word however fails the Hex-Color format validation. Therefore, we had to manually add this into our new

dictionary with the correct formatted colors. The “isrgbcolor()” validates the color, if it is in Hex-Color format. This method returns a Boolean, so when a color is validated, then we can add the color to a new dictionary with the correct color format. The addition of a color is then set by the corresponding year of the website. The color has to be set to the right Hex-Color value format. We then add these newly corrected colors into a newly converted dictionary, (“convertedDict[keyValues].append(values)”) [2][11].

M. Visualization - Getting the Frequency of the Colors

The methods “getFreqColor()” and “newFun()” take in a dictionary that will then be converted into a nested dictionary, with an extra value as the frequency and/or percentage of colors within the corresponding key. In this case the key will be the year of the website's data. The format of the dictionary will look like (“{year : {#color : freq}, {#color2 : freq2}”). The “getFreqColor()” method will simply count the number of occurrences of each color for the associated year. The “newFun()” method will get the percentage of each color from the total of all colors for the specific year. We created these two helper methods for two different visualization outputs. The “getFreqColor()” method is used for output of the JSON file, whereas the “newFun()” method is used for the stacked bar chart. These percentages are used to assign the weights for the stacked bar chart. Both of these methods output a modified version of the nested dictionary in Python. Finally, we use collections from Python to order the dictionary from the latest to earliest years. This makes it easier to set up the graph and also makes a more usable JSON file object [3][12].

N. Visualization - Creating the Stacked Bar Chart

From the nested dictionary, we create a graph using Matplotlib's API. We set the weights with the colors list from the dictionary and also use the percentage to calculate the weight sizes for each bar. Referring to Section *Visualization - Stacked Bar Chart*, each bar of the graph corresponds to the year of the websites. The Y-axis corresponds with the percentage of colors. We create a legend alongside the graph. This is because the graph is slightly larger than intended for the legend. We wanted to make the graph as clear and readable as possible. We use Matplotlib's “savefig” method to make a JPEG image of the graph to output to the user's directory. The method “export_legend” will output a JPEG image of the legend to go alongside with the graph for the state. We designate the name by the state that is used for extraction. We have a list of all US states and territories that will be used to check with the config's file's tourism website URL [11].

O. Visualization - Outputting JSON File

Another part of our visualization is to create a JSON file that shows our nested dictionary. Since dictionaries in Python have a similar format as a JSON object, having a JSON file allows for future research on color extraction. You can use it to plot graphs with just the frequency count of colors rather than percentages. This is something that we can leave with other future groups and research teams to use. We create the JSON file by using the “json.dumps” method in JSON. Using the formation tools with “json.dumps”, you can format the JSON object however you like. The whole method for creating a JSON file is shown by “json.dumps(overallDictionary, sort_keys=True, allow_nan = True, indent=4)”. Like the stacked bar chart and legend for the graph, we also output the JSON object into the same user directory. It is denoted by the corresponding state’s name from the config file [7][12].

VI. Assessment of Implementation

This section will address concerns and possible errors that may be present in the code that should be looked at in the future to determine certainty in the data. This is due to the fact that the data set is extremely large and will be expanded to work on every state's website. We cannot say with clear certainty the code will work without failure. Based on testing and our personal usage the code is working as expected, but may be lacking data or parsing data incorrectly in edge cases. All concerns will be addressed below in the following sections, in addition to addressing concerns with the tech stack.

A. Jupyter Notebook

The use of Jupyter Notebooks was talked about in the implementation section. To summarize, the main reason it was chosen was because the previous group chose it as their development environment, thus to build onto that we maintained usage of the Jupyter Notebook. The following issues were noticed when using Jupyter Notebooks:

- Lack of the ability to use command arguments from Python’s built-in system.args
- Difficulty in using GitLab for version control

Focusing on the first of the two issues listed above, due to how the Jupyter Notebook creates the file and runs the processes, there was no easy way to implement arguments to the main function. Likewise, there was no ability for us to use a terminal to run our code on or a way for us to port our code into a Python file that could be run on the command line. This led to the creation of a configuration file that would input needed parameters for the program. This configuration file is extremely rudimentary, which

could lead to issues if users do not use the file as intended. This process could be worked on further, but due to time constraints and this being a Jupyter Notebook related issue this process was not further refined.

Focusing on the second of the two issues listed above, we noticed due to how Jupyter Notebooks creates the IPYNB file, it includes output with the provided code. So any use of a version control system like GitLab will lead to continuous changes as outputs may be different across different developers' testing purposes. However, this will be shown on the main file as changes. We began using a version control system on Discord by discussing changes in a comment under the file we uploaded manually. This led to some issues relating to Unicode encoding when outputting the CSV file as we utilized two different operating systems: MacOS and Windows. This issue was addressed, but may come up in the future when developing code through this system.

B. Data Extraction Process

The main bulk of the code focusing on data extraction used BeautifulSoup to determine parts wanted from the Parquet files provided. We can say with fair certainty that BeautifulSoup is grabbing the parts we wanted from the respective sources as we designed, but we cannot say with extreme certainty that it is not missing parts from the data. If, for example, whoever encoded the websites for the tourist website did not use normal conventions, BeautifulSoup may miss those data pieces. Likewise, data may be encapsulated in places that are not conventional to normal HTML standards, which means it will not be found in most of our approaches to everything except the extraction of background colors. The extraction of background colors uses a custom in-house parser on the Parquet payload. The reason for this is because we needed to find every instance of "bg-color" and "background-color" (HTML styling names), which can be located in a variety of tags throughout the Parquet payload. The in-house parser purely looks for these two styling arguments. If for some reason these two styling attributes were misspelled or abstracted into a CSS custom styling, then this parser will be unable to determine the background color and will ignore this data. This is purely a concern, but this does not mean it is a problem we have observed ourselves in our usage.

Another concern with the code created is the lack of extensive error checking. If as described above there are HTML conventions that are not standard, our error checking should be able to handle that [10]. However, there is still a possibility that this is not the case. This may lead to major problems in usage of the code, which should be noted and worked on by future teams. In addition, the structure of our code is not abstract. Many things were not created with methods or functions in mind when they should have been. For example, some code is repeated with slight variations, which could have easily

been abstracted into a method with different parameters. Additionally, some variables may be hard to understand as they were quickly switched to adjust for more needed extraction when developing the code. Some parts of the code should be abstracted into methods to help maintainability and iron out possible bugs when expanding the code base. Thus, future groups should work on maintainability and cleaning up the code. Some aspects of the code are abstracted which are mentioned in the implementation section.

The last concern we have is with our custom data parser. Based on the implementation it should be able to handle any work, but due to it not being abstracted into a method it may be hard to use by future groups. Additionally, based on the implementation it should be able to find anything that uses standard HTML conventions, but from usage we have found that not all of the data provided is in this formula, i.e., splicing data based on delimiters [10].

C. Time Complexity

The sheer amount of data has led to realizations that our code could be more efficient in fetching this data. Specifically, the custom data parser we created is extremely inefficient as it parses the whole payload for specific keywords, then outputs those positions to be grabbed by a following process. Based on how BeautifulSoup extracts data it may suffer from this same dilemma [5]. Future groups working on this project should attempt to find a way to simplify the process or link the processes to the config file arguments, so not all of the data is always extracted if it is not necessary. Likewise, the use of the Jupyter Notebook may be affecting the time to complete the application [4]. It seems to create extra steps in the execution of the program through its usage of blocks and data output points.

VII. User's Manual

In order to have a system able to run the code please follow the Developer's Manual installation instructions to set up your Jupyter Notebook [4]. Unfortunately, the code is not developed with its own GUI or compiled into an .exe for easy usage. Future groups may want to look into implementing these features.

After following the Developer's Manual and setting up a workspace with the wanted raw data and file "extractAll_openAllFiles" you are able to begin setting up the code. Two things need to be changed before execution can be done. First the config.txt file must be updated with the wanted website URL or hostname. The example config.txt will already include www.colorado.com, which can be updated to any hostname that another

tourism website may have. Make sure to include the “www.” when inputting this into the config file in addition to the website’s extension (e.g., .com). Do not include anything before the “www.” Next update the directorToChange variable located in block 5 with the directory that contains the Parquet files as shown in Figure 6 [2].

You are now able to run the code by selecting Kernel->Restart & Run All. If an error occurs with the config file, it will be preceded by a print statement denoting a possible error in using the system. Please follow along with those instructions and adjust accordingly.

Our program is divided into two parts, a data extraction portion and a visualization portion. When running the two programs for this application, start by running the extractAll program. Without running extractAll first, the visualization program will not have any data present to be used in constructing the stacked bar chart. The extractAll program will output a folder for each Parquet file. This folder will contain CSV files with their respective data changes. This spring semester, we decided to create a visualization of frequency colors shown on tourism websites. The program, visualizationFirstStage, will output a JSON file (Figure 2), and two JPEG images [2][12].

The two JPEG images will include the stacked bar chart for the color visualization of tourism websites, as well as the legend that supports the chart with the different colors found. Examples of a finalized stacked bar graph can be seen in Figure 3, Figure 4, and Figure 5, and an example of a finished legend JPEG can be seen in Figure 6. The naming format for these files are based on the tourism website’s specific state. The stacked bar chart will be designated by the state name. The naming format for the legend of the char will just start with “legend” followed by the state name.

```

1 {
2   "1996": {
3     "#FFFFFF": 1
4   },
5   "1997": {
6     "#698EFE": 30,
7     "#BDD5FF": 69,
8     "#FFFFFF": 447
9   },
10  "1998": {
11    "#CCCCC": 1,
12    "#F6EED0": 5,
13    "#FFFFFF": 511
14  },
15  "1999": {
16    "#AAAAAA": 9,
17    "#CCCCC": 135,
18    "#DDDDDD": 72,
19    "#F6EED0": 4,
20    "#FFFFFF": 480
21  },
22  "2000": {
23    "#000000": 3,
24    "#AAAAAA": 15,
25    "#CC3300": 1,
26    "#CCCCC": 220,
27    "#DDDDDD": 112,
28    "#F6EED0": 2,
29    "#FFFFFF": 666
30  },
31  "2001": {
32    "#000000": 1,
33    "#999999": 63,
34    "#AAAAAA": 3,
35    "#CC3300": 1,
36    "#CCCCC": 204,
37    "#DDDDDD": 16,
38    "#FFFFFF": 203
39  }
40 }

```

Figure 2: Example of a Completed JSON File

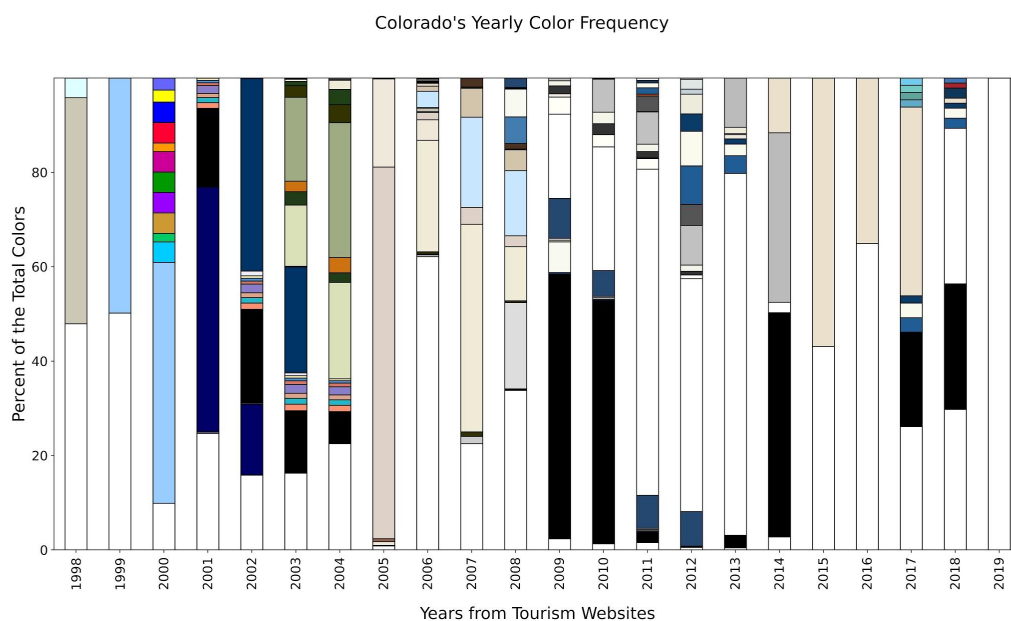


Figure 3: Colorado's Completed Stacked Bar Chart

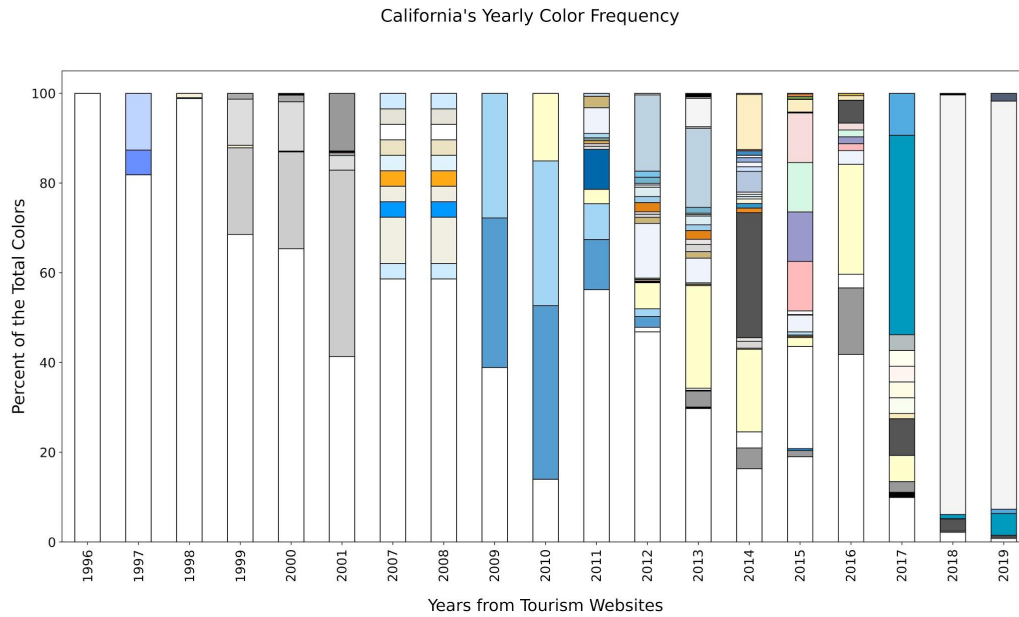


Figure 4: California's Completed Stacked Bar Chart

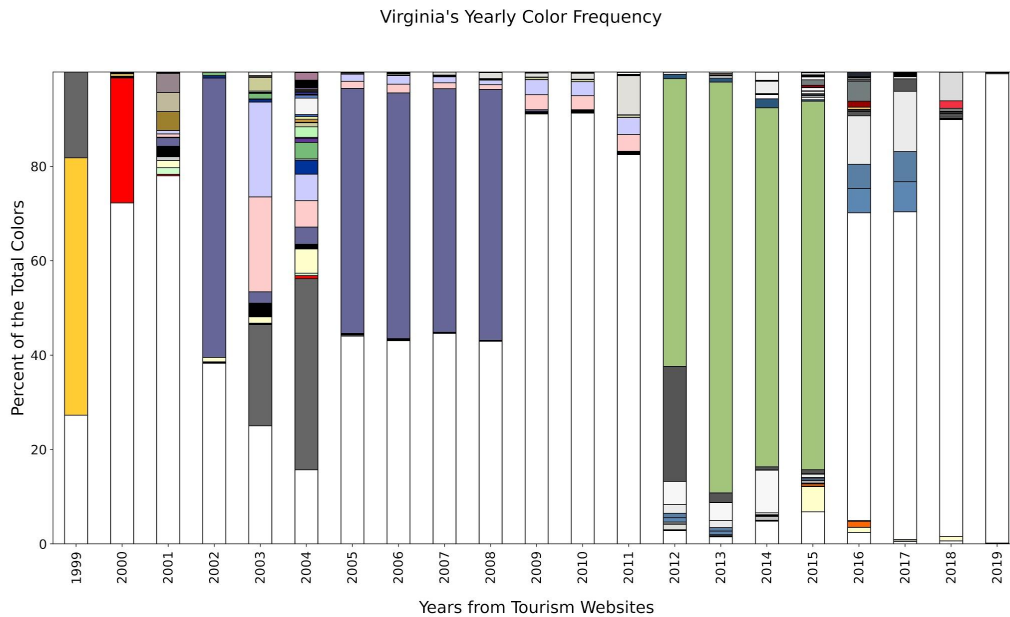


Figure 5: Virginia's Completed Stacked Bar Chart






















































































	#FFFFFF		#F5F5F6
	#698EFE		#FEFEFE
	#BDD5FF		#0F85B5
	#CCCCCC		#FFBBBB
	#F6EED0		#9999CC
	#DDDDDD		#D6F7E4
	#AAAAAA		#F7DCDC
	#CC3300		#555555
	#000000		#ED8B1D
	#999999		#329ACD
	#CDECFE		#FFF6DE
	#F1EEE2		#E0F4F7
	#0099FF		#DFF2FB
	#F0EEE1		#F3F3F3
	#FFAA17		#B5C8E0
	#E1F3FF		#CDE3FF
	#ECE3C2		#F1F3FF
	transparent		#8DB0DE
	#E5E3D8		#F7F7F7
	#CFEBFF		#3F8CC4
	#549DD0		#F68B33
	#A3D6F5		#FDEFC3
	#FFFFCC		#FFD528
	#0067AB		#73AA4E
	#E7E7E8		#F68B34
	#D1D2D4		#FAE8BC
	#E8AE4A		#E3BA12
	#7AA395		#F8FFF0
	#ABDCF6		#FFFCE5
	#EFF3FB		#FEF5F1
	#CBB677		#FFFFFF0
	#BED9ED		#B5BEBE
	#D8D7D5		#009ABF
	#E8E6E6		#53ACE1
	#E48414		#F4F4F4
	#A0CFEB		#008C99
	#DDEEF2		#005966
	#E8E6E7		#DE4826
	#CFE7F5		#AB1500
	#73B5CC		#009DDC
	#7AC5EC		#006AA9
	#8DD3E1		#525D76
	#EAECF9		

Figure 6: Example of California's Legend for the Stacked Bar Chart

Figure 2 shows an example of the JSON file output, from running the visualizer application [12]. The stacked bar chart visualizations in the three states can be used to display the variance of colors in each year (Figure 3, Figure 4, Figure 5). From analyzing the colors in later years, we see that there is a more increase of white and lighter tone colors. Figure 5 shows an example of how a legend for California's stacked bar chart will output. Like the stacked bar chart for each state, the legend is outputted as a JPEG image format.

VIII. Developer's Manual

A. Jupyter Installation

Similar to the Spring 2020 US Tourism group, our installation process is relatively the same. On MacOS, it would be easier to install homebrew [8] to install Jupyter Notebook and the Python subsidiaries as necessary. For Windows users the process begins by installing Anaconda Navigator [6].

B. Library Installation

MacOS

Once you have Jupyter Notebook installed and can open up an actual notebook, you need to install the various libraries that we use. The following libraries are necessary: pyarrow, Pandas, BeautifulSoup [3][5][9]. Install these in your coding environment by doing the following commands:

```
pip install pyarrow
pip install pandas
pip install bs4
```

If you have Python version 3 or newer, do the following:

```
pip3 install pyarrow
pip3 install pandas
pip3 install bs4
```

Your Jupyter Notebook should say these libraries have been installed as shown in Figure 7. In order to run the cells, click on restart and run all as shown in Figure 8.

Windows

Once Anaconda Navigator is installed, go ahead and launch Jupyter Notebook, which should open a web directory on your C drive [4][6]. Navigate to where your workspace is that should include any raw data and our provided code, for example, Desktop->CS4264-Workspace. Once here, click the New Button located at the top right and select Python 3. A file should be created titled Untitled.ipynb. Click on this file and a blank cell should be seen. In the cell, input as follows:

`pip install pyarrow`

Then click Kernel->Restart & Run All. Now restart Jupyter Notebook. Then navigate back to your workspace. You should be able to run the provided code we created [4].

In order to run your cells in Jupyter Notebook, click on Kernel and then Restart & Run All as shown in Figure 8. Figure 7 shows how everything should appear after you have finished installing. You can also install these dependencies on your command prompt, however we recommend installing on the Notebook environment. We found that installing libraries on the terminal leads to the issue of having to always reinstall the same libraries every time one closes/opens up a new session of Jupyter Notebook [4].

```
In [1]: pip install pyarrow
```

```
Requirement already satisfied: pyarrow in /usr/local/Cellar/jupyterlab/3.0.9/libexec/lib/python3.9/site-packages (3.0.0)
Requirement already satisfied: numpy>=1.16.6 in /usr/local/Cellar/jupyterlab/3.0.9/libexec/lib/python3.9/site-packages (from pyarrow) (1.20.1)
Note: you may need to restart the kernel to use updated packages.
```

```
In [2]: pip install pandas
```

```
Requirement already satisfied: pandas in /usr/local/Cellar/jupyterlab/3.0.9/libexec/lib/python3.9/site-packages (1.2.3)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/Cellar/jupyterlab/3.0.9/libexec/lib/python3.9/site-packages (from pandas) (2.8.1)
Requirement already satisfied: numpy>=1.16.5 in /usr/local/Cellar/jupyterlab/3.0.9/libexec/lib/python3.9/site-packages (from pandas) (1.20.1)
Requirement already satisfied: pytz>=2017.3 in /usr/local/Cellar/jupyterlab/3.0.9/libexec/lib/python3.9/site-packages (from pandas) (2021.1)
Requirement already satisfied: six>=1.5 in /usr/local/Cellar/jupyterlab/3.0.9/libexec/lib/python3.9/site-packages (from python-dateutil>=2.7.3->pandas) (1.15.0)
Note: you may need to restart the kernel to use updated packages.
```

```
In [3]: pip install bs4
```

```
Requirement already satisfied: bs4 in /usr/local/Cellar/jupyterlab/3.0.9/libexec/lib/python3.9/site-packages (0.0.1)
Requirement already satisfied: beautifulsoup4 in /usr/local/Cellar/jupyterlab/3.0.9/libexec/lib/python3.9/site-packages (from bs4) (4.9.3)
Requirement already satisfied: soupsieve>1.2 in /usr/local/Cellar/jupyterlab/3.0.9/libexec/lib/python3.9/site-packages (from beautifulsoup4->bs4) (2.2)
Note: you may need to restart the kernel to use updated packages.
```

Figure 7: Installing the required libraries

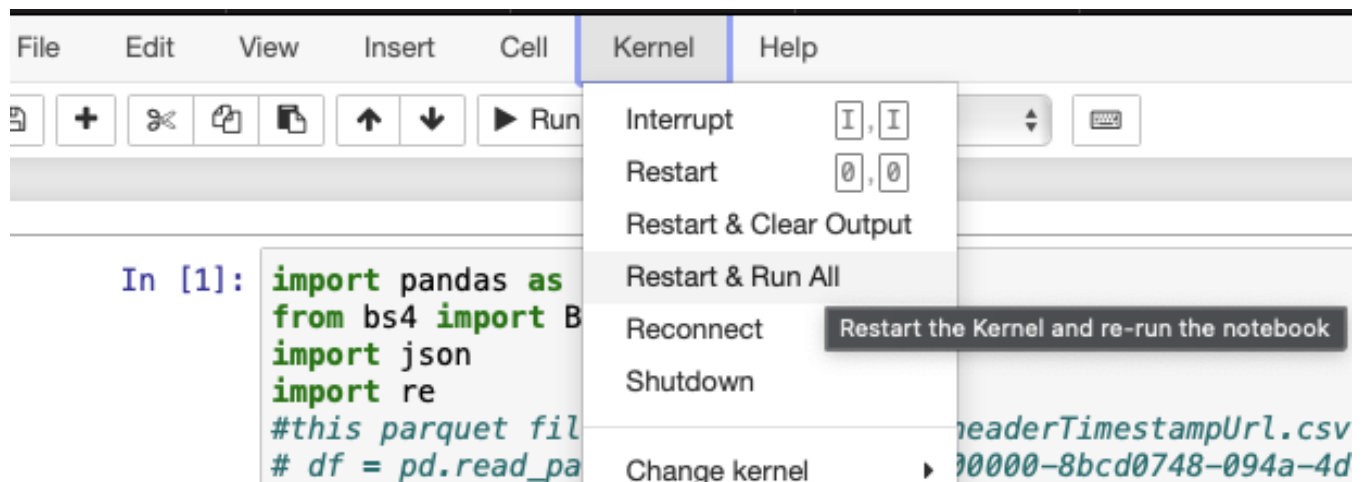


Figure 8: Running the cells in Jupyter Notebook

C. Loading files, Extraction, and CSV output

The Jupyter notebook file (.ipynb extension) called `extractAll_openAllFiles.ipynb` contains all our code for loading all files, extraction, and CSV output. The notebook starts off with our `headerAndTime(df,webName)` as shown in Figure 9 [4]. This method handles all our extraction (please see the implementation section to understand how we extract the various data). We've left each different extraction process in each of its own comment blocks so one can easily identify what extraction part they are looking at.

```
[3]: def headerAndTime(df, webName):
    #list that will hold information for every row
    imageInfo = []
    #list that will hold indices of rows with errors
    errors = []
    #this will hold information for every raw text row
    rawTextInfo = []
    #this will hold external link info
    externalInfo = []
    #this will hold color info
    colorInfo = []
    #this will hold meta tag info
    metaInfo = []

    filename = ''
    rawtxt_fname = ''
    websiteName = webName
    foundName = False

    oldest = 1
    youngest = 99999999
    for row in range(df.shape[0]):
        try:
            #dictionary to store information from the row
            imageDict = {}
            rawTxtDict = {}
            linkDict = {}
            colorDict = {}
            metaDict = {}
            #grabbing information from each column of the row
            soup = BeautifulSoup(df.payload[row], 'html.parser')

            #
            bodycolor = soup.find_all('body', style=True)
            for bcolor in bodycolor:
                print(bcolor['bgcolor'])
                print(bcolor['style'])
            #

            #-----Grabbing all External links-----
            linkDict["Timestamp"] = df.timestamp[row]
            linkDict["URL"] = df.originalUrl[row]
```

Figure 9: Our `headerAndTime()` method

`loadFile(pathDir)` in our notebook handles the logic for reading in all Parquet files shown in Figure 10. Our code works so that users can store all their Parquet files into a folder and just provide the path to the folder (in the `directoryToChange` variable as seen in Figure 11) [2]. You can also add the host name of the website in the configuration file. Changing the file path will need to be manually updated in this method. Replace the 'directoryToChange' variable that is already present.

```
# this will load all the parquet files in the set directory
def loadFile(pathDir): #pass file path to your directory
    list_files = []
    os.chdir(pathDir)

    for file in glob.glob("*.snappy.parquet"):
        # Dont read in the directories, only the files
        if "output_" in file:
            continue
        else:
            list_files.append(file)
    return list_files

#-----
dir = directoryToChange
#-----
os.chdir(dir)
log_file = open("log.txt", "w+") #handles log file

f = loadFile(dir)

size_list = len(f)
counter = 1
```

Figure 10: `loadFile()` and reading in all Parquet file

```
# CHANGE THIS TO YOUR DIRECTORY
#-----
# Example File path in Mac: directoryToChange = "/Users/abhinavverelly/Desktop/Hypertext_Project/try/test"
directoryToChange = "/Users/abhinavverelly/Desktop/Hypertext_Project/vatemp"
# Make sure to change directoryToChange variable with the correct file path
#-----
# CHANGE THIS TO YOUR DIRECTORY
```

Figure 11: `directoryToChange` variable that users need to change

We also handle output in an easier to understand and efficient manner. We output each CSV into a unique folder with the same name as the Parquet file. We handle this part of the code in the loop that deals with reading in the Parquet file shown in Figure 12. We also print out the directory each time as we run the code for debugging purposes to ensure we are outputting each CSV to the correct directory. You can refer to this image for how the folders look once the Parquet files have been read (Figure 13). How the CSVs will look inside each of the unique directories is shown in Figure 14 [2].

```
#makes a new directory for all the outputed csv files
#this is the output_ folowed by the name of the parquet file
os.mkdir(dir + "/" + "output_" + f[k])
#add csv files to new directory
os.chdir(dir + "/" + "output_" + f[k])
print(os.getcwd())
```

Figure 12: Outputs each CSV to unique folder

Name	Date Modified	Size	Kind
part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c000.snappy.parquet	Jan 7, 2021 at 12:19 AM	210.2 MB	Document
part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c002.snappy.parquet	Jan 7, 2021 at 12:20 AM	190.9 MB	Document
part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c001.snappy.parquet	Jan 7, 2021 at 12:19 AM	189.1 MB	Document
part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c005.snappy.parquet	Jan 7, 2021 at 12:20 AM	170.1 MB	Document
part-00000-6b7a9564-344f-474c-a7dd-71b49ecb3fde-c000.snappy.parquet	Feb 13, 2021 at 2:18 PM	163.9 MB	Document
part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c004.snappy.parquet	Jan 7, 2021 at 12:20 AM	155.2 MB	Document
part-00000-e16e0ba2-f610-416d-8134-0c5bd99ff962-c000.snappy.parquet	Jan 15, 2021 at 8:53 PM	129.9 MB	Document
part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c003.snappy.parquet	Jan 7, 2021 at 12:20 AM	105.8 MB	Document
part-00000-e16e0ba2-f610-416d-8134-0c5bd99ff962-c015.snappy.parquet	Feb 22, 2021 at 11:58 AM	35.4 MB	Document
config.txt	Mar 10, 2021 at 9:43 PM	265 bytes	Plain Text
output_part-00000-6b7a9564-344f-474c-a7dd-71b49ecb3fde-c000.snappy.parquet	Mar 31, 2021 at 4:29 PM	--	Folder
output_part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c000.snappy.parquet	Mar 31, 2021 at 4:35 PM	--	Folder
output_part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c001.snappy.parquet	Mar 31, 2021 at 4:25 PM	--	Folder
output_part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c002.snappy.parquet	Mar 31, 2021 at 4:08 PM	--	Folder
output_part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c003.snappy.parquet	Mar 31, 2021 at 4:19 PM	--	Folder
output_part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c004.snappy.parquet	Mar 31, 2021 at 4:16 PM	--	Folder
output_part-00000-97f6d37e-0a9d-4f9f-a313-5097fa75bb58-c005.snappy.parquet	Mar 31, 2021 at 4:12 PM	--	Folder
output_part-00000-e16e0ba2-f610-416d-8134-0c5bd99ff962-c000.snappy.parquet	Mar 31, 2021 at 4:01 PM	--	Folder
output_part-00000-e16e0ba2-f610-416d-8134-0c5bd99ff962-c015.snappy.parquet	Mar 31, 2021 at 4:02 PM	--	Folder

Figure 13: Output of each Parquet file into unique directories

output_part-00000-6b7a9564-344f-474c-a7dd-71b49ecb3fde-c000.snappy.parquet				
	Name	Date Modified	Size	Kind
	19980423-201....com_Colors.csv	Mar 31, 2021 at 4:29 PM	847 KB	Comm...et (.csv)
	19980423-201....m_ExtIntLink.csv	Mar 31, 2021 at 4:29 PM	5.6 MB	Comm...et (.csv)
	19980423-201....com_Images.csv	Mar 31, 2021 at 4:29 PM	3.2 MB	Comm...et (.csv)
	19980423-201....o.com_Meta.csv	Mar 31, 2021 at 4:29 PM	1.3 MB	Comm...et (.csv)
	19980423-201....com_RawTxt.csv	Mar 31, 2021 at 4:29 PM	40.8 MB	Comm...et (.csv)

Figure 14: example of contents in output folder

Moving onto the CSV output portion, this part is handled by the `makeCSV(filename, columns_name, tableInfo)` method. Without repeating the same information in the implementation section, this method makes it flexible so any new data extraction that can be done in the future, can simply use this method in order to create a CSV file. Figure 15 shows our `makeCSV()` method as well as how we use it.

```

#method that will take in filename, columns that will showup in CSV, and every extracti
#using basic way of making a csv file in python to spit out a .csv file
def makeCSV(filename, columns_name, tableInfo):
    try:
        with open(filename, 'w', newline='', encoding="utf-8") as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=columns_name)
            writer.writeheader()
            for data in tableInfo:
                writer.writerow(data)
    except IOError:
        print("I/O error")

#Headers for each CSV file
csv_image_columns = ['Timestamp', 'URL', 'Number_Trackers', 'Number_Images', 'Imag
rawtxt_csv_columns = ['Timestamp', 'URL', 'Number_Words', 'Raw']
externalLink_csv_columns = ['Timestamp', 'URL', 'Number_ExternalLinks', 'Externall
externalLink_csv_columns = ['Timestamp', 'URL', 'Number_ExternalLinks', 'Externall
color_csv_columns = ['Timestamp', 'URL', 'Number_Colors', "Colors", 'Number_Backgr
meta_csv_columns = ['Timestamp', 'URL', 'Number_Descrp Tags', 'Meta Description', '

#file name for each csv file
csv_image_file = filename
raw_file = rawtxt_fname
link_file = extlink_fname
color_file = color_fname
meta_file = meta_fname

#makes a new directory for all the outputed csv files
#this is the output_ followed by the name of the parquet file
os.mkdir(dir + "/" + "output_" + f[k])
#add csv files to new directory
os.chdir(dir + "/" + "output_" + f[k])
print(os.getcwd())

#spits out CSV for colors[bgcolor, background-color]
print("printing color")
makeCSV(color_file, color_csv_columns, colorInfo)

print("printing image")
#spits out CSV for images
makeCSV(csv_image_file, csv_image_columns, imageInfo)

print("printing raw")
#spits out CSV for raw text
makeCSV(raw_file, rawtxt_csv_columns, rawTextInfo)

print("printing link")
#spits out CSV for external links
makeCSV(link_file, externalLink_csv_columns, externalInfo)

print("printing meta")
#spits out CSV for meta tags
makeCSV(meta_file, meta_csv_columns, metaInfo)
#set back default directory
os.chdir(dir)
print(os.getcwd())
print("csv created: " + str(counter))
counter = counter + 1

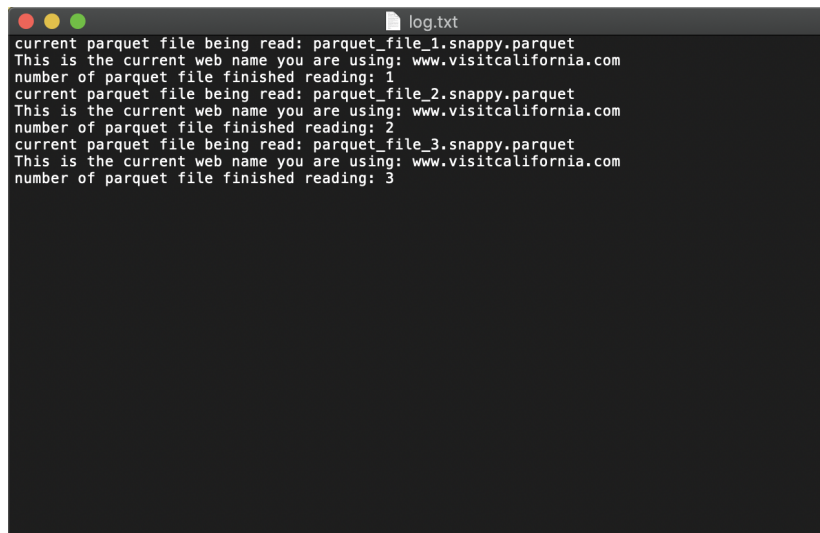
```

Figure 15: makeCSV method and its usage

In terms of timing for the output, we loaded and read in eight sample Parquet files (for the state of Colorado). Each Parquet file yields a CSV file (in our case we output 5 CSV files: colors, images, links, raw text, meta tags), and the total size of the 8 Parquet files came out to be 1.3505gb. It took 41 minutes to finish reading and outputting the CSV contents for all of the Parquet files. Our total time for extractions for the other states as well as number of Parquet files can be seen in Table 2. We also output a log file that will keep track of what Parquet files have been read, along with a counter of the number of Parquet files read (Figure 16) [2].

State	Number Of Parquet File	Total File size	Total Time For Extraction
Colorado (Not all files)	8	1.3505 GB	~41 minutes
Colorado	23	2.43 GB	~130 minutes
California	14	3.49 GB	~202 minutes
Virginia	160	7.47 GB	~270 minutes

Table 2: Total Time For Extraction



```

log.txt
current parquet file being read: parquet_file_1.snappy.parquet
This is the current web name you are using: www.visitcalifornia.com
number of parquet file finished reading: 1
current parquet file being read: parquet_file_2.snappy.parquet
This is the current web name you are using: www.visitcalifornia.com
number of parquet file finished reading: 2
current parquet file being read: parquet_file_3.snappy.parquet
This is the current web name you are using: www.visitcalifornia.com
number of parquet file finished reading: 3

```

Figure 16: Sample output of log.txt

D. Handling Visualization - Extracting Color CSV Files and Storing

We read the color CSV files, by going through and finding all the output directories. This is achieved by the function `loadFile()` in Figure 17. We then are able to open the directories and find the color CSV files. This is then used for the overall visualization. Figure 18 goes through opening these output directories and grabbing the colors and corresponding websites years to put into a dictionary in Python [7].

```

59
60 # This will load all the parquet files in the set directory
61 def loadFile(pathDir): #pass file path to your directory
62     list_files = []
63     os.chdir(pathDir)
64     for file in glob.glob("*.snappy.parquet"):
65         print("this is the file name: ", file)
66         list_files.append(file)
67     return list_files
68

```

Figure 17: Load all Output Directories

```

166 # -----
167 dir = directoryToChange
168 # -----
169 counterFile = 0
170 f = loadFile(dir)
171
172 #grabes all CSV files that are _Colors.csv
173 for elem in range(len(f)):
174     output = 'output'
175     if output in f[elem]:
176         os.chdir(dir + "/" + f[elem])
177         # finds all the colors.csv files and get needed data
178         for file in glob.glob("*Colors*.csv"):
179             with open(file) as csv_file:
180                 csv_reader = csv.DictReader(csv_file)
181                 line_count = 0
182                 print("File name being read: ", file)
183                 counterFile = counterFile + 1
184                 print(counterFile, " file completed out of ", len(glob.glob("*.csv")))
185                 for row in csv_reader:
186                     cacluete(row["Timestamp"], row["Colors"])
187 # create a nested dictiaonry to get the year and associated color and frequency of the color
188 colorConvert_Frequency(yearColorListDict)
189

```

Figure 18: Grabbing data from color CSV Files


```

97 #will go thorough each year's color
98 #From there will ensure that all colors grab will be converted to hex format
99 def colorConvert_Frequency(cDict): #
100     for keyValues in cDict:
101         valueList = cDict.get(keyValues)
102         #grab each year from the dictionary passed from parameter and add to new converted dictionary
103         #converted dict holds year(key) : color(now in all hex): freq
104         for values in valueList:
105             # color names be converted to HEX
106             if convertedDict.get(keyValues) == None:
107                 convertedDict[keyValues] = []
108             try:
109                 values = values.lower()
110                 matColor = mat.colors.cnames[values]
111                 convertedDict[keyValues].append(matColor)
112                 continue
113             except:
114                 #handling futher filtrations-----
115                 if len(values) > 0:
116                     values = values.strip()
117                     values = values.upper()
118                     # deal with empty values
119                     if len(values) > 2:
120
121                         if values[0] != "#": # check if parsed values have no # symbol
122                             if values == "LACK" or values == "HITE":
123                                 convertedDict[keyValues].append(mat.colors.cnames["black"])
124                                 continue
125                             values = "#" + values
126                             #regex stuff
127                             if isrgbcolor(values) == True:
128                                 if checkSize(values):
129                                     convertedDict[keyValues].append(values)
130                             else:
131                                 # fix transparent issue
132                                 if "TRANSPARENT" in values:
133                                     values = "transparent"
134                                     convertedDict[keyValues].append(values)
135                                 # fix caps issues
136                                 if "WHITE" in values or "BLACK" in values:
137                                     values = values.lower()[1:len(values)]
138                                     values = mat.colors.cnames[values]
139                                     if checkSize(values):
140                                         convertedDict[keyValues].append(values)
141                                 # get bgcolor tags in old websites
142                                 if isrgbcolor(values) == True:
143                                     values = values[countString(values)-1:countString(values)+7].strip()
144                                     if checkSize(values):
145                                         convertedDict[keyValues].append(values)
146                             else:
147                                 if isrgbcolor(values):
148                                     if checkSize(values):
149                                         convertedDict[keyValues].append(values)
150                             else:
151                                 # check if pattern still exists after all other checks
152                                 if isrgbcolor(values):
153                                     values = values[countString(values)-1:countString(values)+7].strip()
154                                     if checkSize(values):
155                                         convertedDict[keyValues].append(values)
156 # method counts the frequency of colors

```

Figure 19: Checking the colors extracted and validating them

```

90
91 # check if valid HEX Color value
92 def isrgbcolor(value):
93     _rgbstring = re.compile(r'#[a-fA-F0-9]{6}$')
94     return bool(_rgbstring.match(value))
95
96

```

Figure 20: Validating a color into Hex-Color format

Figure 19 goes in depth into the “colorConvert_Frequency()” method. This method goes through the colors of the dictionary’s value to check if it needs to be in Hex-Color format. Figure 20 shows the Hex-Color validation method. We use this to validate the color if it is in the correct Hex-Color format. If this method is true, we can add it to a newly modified dictionary to hold the new color values with the corresponding years.

```

156 # method counts the frequency of colors
157 def getFreqColor(aDict):
158     wordfreq = {}
159     masterDict = {}
160     for key in aDict:
161         for value in aDict.get(key):
162             wordfreq[value] = wordfreq.setdefault(value, 0) + 1
163         masterDict[key] = wordfreq
164         wordfreq = {}
165     return masterDict
166

```

Figure 21: Gets the frequency of the colors in a year

```

193
194 #orders the dictionary so that all the years(key) are sorted in order
195 od = collections.OrderedDict(sorted(getFreqColor(convertedDict).items()))
196 colorfreqfam = []
197 percentDict = {}
198 #will go through the sorted dictionary and change the frequency count to a percentage
199 #will return a ordered(by year) nest dict that holds each color for every year, and each color's frequency percentage
200 def newFun(Dict):
201     count = 0
202     colorPercent = {}
203     for key in Dict:
204         #go through each color and get the frequency of each color for each year
205         #then will count the total frequency of all colors in each year
206         for value in Dict.get(key):
207             colorfreqfam.append(value)
208             freqval = Dict[key][value]
209             count = freqval + count
210         finalCount = count
211         #will make the frequency for each unique color as a percentage
212         for i in Dict.get(key):
213             val2 = Dict[key][i]
214             weight = (val2 / finalCount) * 100
215             colorPercent[i] = weight
216         count = 0
217
218         percentDict[key] = colorPercent
219         colorPercent = {}
220     return percentDict
221

```

Figure 22: Gets the percentage of colors out of the total colors in a year.

Figure 21 will calculate the frequency of all the color values in a dictionary by using Python’s “setdefault” method in a dictionary [7]. This method, “getFreqColor()”, returns a new nested dictionary with the format, (“{year : {#color : freq}, {#color2 : freq2}”). Figure 22 also returns a nested dictionary. However the frequency of color is replaced by the total percentage of each color from each year. The color frequency will be divided by the total number of color frequencies found for each year. The dictionary format will be the same as Figure 21, but will have percentages, (“{year : {#color : percent1}, {#color2 : percent2}”). Figure 21 is used for the output of a JSON File with the same format, whereas Figure 22 is used to visualize the stacked bar chart [7][12]. The nested dictionaries created are also sorted from oldest to newest by year.

E. Handling Visualization: Making the Stacked Bar Chart and JSON File

```
272
273 import json
274 # method to dump total dictionary to json
275 # change directory to the root
276
277 os.chdir(dir)
278 overallDictionary = od
279 # this will dump the dictionary into a json file in the desired specification
280 json_string = json.dumps(overallDictionary, sort_keys=True, allow_nan=True, indent=4)
281 print("json object preview Down Below")
282 print(json_string)
283 with open(getStateName() + ".json","w") as f:
284     f.write(json_string)
285     print("printed json file in directory: ", os.getcwd())
286     f.close()
287
288
289
```

Figure 23: Outputs a JSON File of the frequency type nested dictionary

```
293 # This method will output the legend separately from the graph
294 def export_legend(legend, filename="legend_from_" + getStateName() + ".jpeg"):
295     fig = legend.figure
296     fig.canvas.draw()
297     bbox = legend.get_window_extent().transformed(fig.dpi_scale_trans.inverted())
298     fig.savefig(filename, dpi="figure", bbox_inches=bbox)
299
300
301 # In[11]:
302
303
304 import numpy as np
305 import matplotlib.pyplot as plt
306
307 # set value of points of interest from going through our dictionary object
308 val = newFun(od)
309 data = val.values()
310 cList = convertColorList(val)
311 test_list = list(set(cList))
312 colorDict = makeColorDict(test_list)
313
314
315 # set the plot and make the graph
316 index = pd.Index(val, name='Years from Tourism Websites')
317 df = pd.DataFrame(data, index=index)
318 ax = df.plot(kind='bar', stacked=True, figsize=(20, 10), edgecolor='black', color=colorDict)
319 plt.xlabel('Years from Tourism Websites', fontsize=19, labelpad=20)
320 plt.ylabel('Percent of the Total Colors', fontsize=19)
321 plt.xticks(fontsize=15)
322 plt.yticks(fontsize=15)
323 os.chdir(dir)
324 # make the title
325 plt.suptitle(getStateName() + '\s Yearly Color Frequency', fontsize=20)
326 # create legend, (Maybe useful)
327 # plt.legend(loc="right", bbox_to_anchor=(1.35, 0.5), ncol=2)
328 export_legend(plt.legend(loc="right", bbox_to_anchor=(1.35, 0.5), ncol=2))
329 # add jpeg file to the main point of the directory
330 plt.savefig(getStateName() + '.jpeg', dpi = (500))
331 plt.show()
332
333
```

Figure 24: Outputs a stacked bar chart and legend corresponding to the graph.

The code shown in Figure 23 will output a JSON file of the nested dictionary with the frequency of colors. It had been indented and formatted to allow for a clear and readable JSON file [12]. This file can be used to plot different types of graphs if needed by future groups and research teams. The code shown in Figure 24 will create a stacked bar chart that will help visualize the color extraction data. Examples of how a finished stacked bar chart appears can be seen in Figure 3, Figure 4, and Figure 5 in the User Manual section. Figure 24 also shows how to convert the graph into a separate JPEG image file. The method “savefig()” on line 330 does just that. We also create a legend for the graph and output it as a separate JPEG image for convenience. This legend will include the colors used from the dictionary to show the different color weights for each bar in the graph.

```

252 # Gets the state name from config.txt file
253 def getStateName():
254     os.chdir(dir)
255
256     state_names = ["Alaska", "Alabama", "Arkansas", "American Samoa", "Arizona", "California", "Colorado",
257 file = open("config.txt", "r")
258 for p in range(0, 5):
259     file.readline()
260 webName = file.readline()
261 sA = webName.replace("www.", "")
262 sB = sA.replace(".com", "") or sA.replace(".org", "") or sA.replace(".edu", "")
263 for i in state_names:
264     if i.lower() in sB:
265         return i

```

Figure 25: Gets the state name from the config file

Figure 25 shows how the file designation for the Stacked bar chart, the legend for the chart, and the JSON file are produced [12]. They are named by the state they correspond with, in the config file from data extraction. The “getStateName()” method will read through the config file and compare the state from the tourism website with the list of all United State’s states and territories. This allows for the files to be uniquely named and easy to find and understand.

IX. Methodology

As a group, we were tasked to complete a Methodology assignment to better grasp the problem at hand. This section describes how we dissected the problem down to something more approachable. It allowed us to understand the scope of the problem. In addition, it allowed us to organize our development efforts in a more concise manner. Likewise, it allowed us to show our thoughts with our client, Dr. Florian Zach. We have three sections which each accomplish different things. Our first section, goal of our users, will describe the goals for each type of user for our system. The second section, subtasks of our goals, breaks down each goal into a more simplified and manageable

set of tasks and subtasks. The third section, implementation based services, we made a table to showcase how each task is implemented. The fourth section, Workflows, uses the third section's table to create a list of workflows. These workflows will then cover each service denoted by the table for each goal. We also have a visual of the workflow to show the system.

A. Goals of our Users

- a. **Data Extractor** - The ability to use the system to parse through different types of HTML data to extract the wanted information.
- b. **Data Visualizer** - The ability to use the data extracted to create visualizations of the data based on various groupings or subsections.

B. Subtasks of our goals

- a. **Extract Data** - Be able to view the data in a readable fashion to be worked on
 - i. Parse through raw HTML to extract necessary components
 - A. Extract external/internal link structure
 - B. Extract meta tags
 - C. Extract raw text
 - a. Determine word count
 - D. Extract Images
 - a. Determine image count
 - b. Determine Tracker count
 - E. Extract color schemes ->
 - a. Grab Background images
 - b. Grab Background-color and bg-color colors
 - F. Output a log file with contents of extraction for tracking purposes
 - ii. Scale extraction process + Simplify Time Complexity
 - A. Extract through each state
 - B. Allow all Parquet to be read from a directory instead of manually and output all files in unique directory
- b. **Visualize Color Data Comparison** - Compare color schemes of different tourist websites

- i. Visualization: Output a stacked bar chart to represent comparison of colors
 - A. Output a JPEG version of the stacked bar chart.
 - B. Output a legend for the map with color names for the different weights of the stacked bar chart.
- ii. Output JSON for nested dictionary of {year: {color_name: fr}}

C. Implementation-based services

Service ID	Service Name	Input file name(s)	Input file Ids (comma-sep)	Output file name	Output file ID	Libraries; Functions; Environments	API endpoint (if applicable)
Extract External / Internal Links	Extract Data: Go through Parquet file and grab all Internal links + external links. Determine how many external links there are and how many internal links there are	Parquet File	1	_ExtIntLinks.csv	2	Jupyter Notebook for coding environment, Pandas to read in Parquet files, BeautifulSoup to parse through HTML, CSV to output files as CSV	N/A
Extract Meta tags	Extract Data: Go through Parquet files and grab Meta Tags. From the meta tags, filter out the content associated with name="description" and name="keywords"	Parquet File	1	_Meta.csv	3	Jupyter Notebook for coding environment, Pandas to read in Parquet files, BeautifulSoup to parse through HTML, CSV to output files as CSV	N/A
Extract Raw text	Extract Data: Go through Parquet file and grab all of Raw Text and determine word count. When determining word count, exclude any of these characters: ,;:><»>><<.../ + &©/	Parquet File	1	_RawTxt.csv	4	Jupyter Notebook for coding environment, Pandas to read in Parquet files, BeautifulSoup to parse through HTML, CSV to output files as CSV	N/A

Extract Images	Extract Data: Go through Parquet files and grab all Images (use any tags that include format src=, in order to not exclude any images). There are also trackers that are embedded into images for AD purposes, grab frequency of those as well.	Parquet File	1	_Images.csv	5	Jupyter Notebook for coding environment, Pandas to read in Parquet files, BeautifulSoup to parse through HTML, CSV to output files as CSV	N/A
Extract Colors	Extract Data: Go through Colors	Parquet File	1	_Colors.csv	6	Jupyter Notebook for coding environment, Pandas to read in Parquet files, BeautifulSoup to parse through HTML, CSV to output files as CSV	N/A
Log File	Log File: Will showcase which Parquet file has been read, how many files has been read for tracking purposes especially with larger data sets	N/A	N/A	log.txt	7	Built in Python functions to read and write out a file	N/A
Read in other state Parquet file	Scale Extraction: Read in Parquet files for different states (we are given 3 but we were working on one state's Parquet file)	Parquet File	1	N/A	N/A	Jupyter Notebook for coding environment, , Pandas to read in Parquet files, BeautifulSoup to parse through HTML, CSV to output files as CSV	N/A
Read in multiple Parquet files at once	Scale Extraction: Read in all Parquet files from one directory and output each file into its own unique directory	All Parquet Files	1	N/A	N/A	Jupyter Notebooks for coding environment, glob and os to read in all files in a directory and output all files in each unique directory	N/A

Visualiz e Colors	Visualization: Compare color schemes of different tourist websites with a stacked bar chart	All_Color.csv output file for each Parquet file	6 (referring to all the Color CSVs we output)	_colorNestedDict.js on legend_from_{state} .jpeg {sate}.jpeg	10,11 and 12 (Referr ing to the JSON and image + legend of the bar graphs)	Jupyter Notebooks for coding environment, glob and os to read in all Colors CSV files from all directories, Pandas used for index data for our graph, collections used for sorting dictionaries, matplotlib for actually plotting our data, json for outputting our nested dictionary as a json object.	N/A
-------------------------	---	--	--	--	--	---	-----

Table 3: Implementation Service Table

Note - {state} stands for visualization of respective state names.

D. Workflows

a. Workflow #1:

User → Goal 1 → Workflow 1

Workflow 1 = Service 1A + Service 1B + Service 1C + Service 1D +
Service 1E + Service 1F:

Service 1A: Extract Data: Go through Parquet file and grab all Internal
links + external links. Determine how many external links there are and
how many internal links there are

Service 1B: Extract Data: Go through Parquet files and grab Meta Tags.
From the meta tags, filter out the content associated with
name="description" and name="keywords"

Service 1C: Extract Data: Go through Parquet file and grab all of Raw Text
and determine word count. When determining word count, exclude any of
these characters: ,;:><»>><<.../ +|&©/

Service 1D: Extract Data: Go through Parquet files and grab all Images (use any tags that include format src=, in order to not exclude any images). There are also trackers that are embedded into images for AD purposes, grab frequency of those as well.

Service 1E: Extract Data: Go through Colors

Service 1F: Export the data into CSV files

b. Workflow #2

User → Goal 2 → Workflow 2

Workflow 2 = Service 2A + Service 2B + Service 2C

Service 2A: Visualization: Output JSON object from the color dictionary

Service 2B: Visualization: Output a Jpeg file of the Stacked bar chart from Jupyter Notebook.

Service 2C: Visualization: Output a Jpeg file of a legend of the colors, from the extraction, from the Stacked Bar chart.

c. Workflow Diagram Below

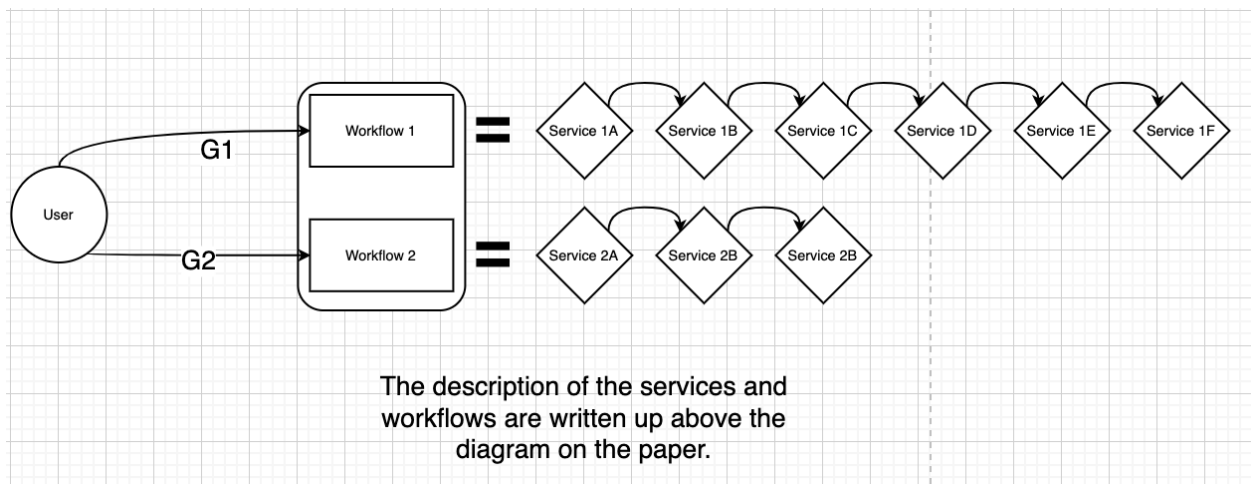


Figure 26: Workflow Image

X. Lessons Learned

A. Timeline / Schedule

Date	Event
3-Feb	Client Meeting - Discuss research goals for upcoming milestone
9-Feb	Flush out research goal / Finalize tools and technology / Flush out deliverable
2/9 - 2/14	Research Period for the project with scope set
15-Feb	Client Meeting - Meeting in regards to current work
2/15 - 2/22	Began extraction of raw text and external links
22-Feb	Client Meeting - Updates / Discuss wanted format of extracted data
2/23 - 2/28	Extraction for Raw Text completed. Working on other parts of the extraction.
28-Feb	Presentation 1 - Project Description / Requirements / Planning
2/28 - 3/8	Extraction for External Links completed.
8-Mar	Client Meeting -Showcase POC / Discuss wanted Visualization / Comparisons
9-Mar	Extract -> raw text / content length / meta tags / file formats / Colors and Background-color / Images
3/9 - 3/22	Finish extracting colors and images. Raw text, Meta tags, and External/Internal Links extraction completed.
22-Mar	Client Meeting - Updates / Showcase of our extraction / mock-up visualization design and implementation from POC
23-Mar	Extraction of all needed elements completed, upon clients request / Created multi-file processor / scope set for scalability and visualization
1-Apr	Presentation 2 - Project design / Initial Implementation / Plan
26-Apr	Client Meeting - Updates / Followup / Extension of Milestone
2-May	Final Presentation
5-May	Project completed and submitted all documentation needed for Class

Table 4: Project Schedule

B. Problems

One of the problems that we faced had to do with the continued issue of Covid-19 and with online classes. Doing such a large project and not being able to meet in person caused issues in terms of performing consistent work throughout each week.

Another problem we faced was the large learning curve of understanding our tools. None of our team members had used Jupyter Notebook. Although we were familiar with Python, parsing HTML from a Parquet file was a new area for us [2][10]. Just getting to open up a Parquet file was a challenge in itself. The preexisting code base from May 2020's team gave us a starting point, but lack of clear comments as well as overall design choice stalled us for a bit. Having to manually paste in the name of the Parquet file confused us as a design choice and took time away from us extracting to make reading in Parquet files abstract. The overall wait time needed to read in a file, extract, and output all the CSVs was a minimum of 30 minutes (or more). This wait time increased with the addition of more Parquet files, as well as larger sized Parquet files [2].

C. Solutions

We dealt with working around Covid-19, by meeting up with the group on a weekly basis. We met with our client every other week to show our overall progress of the project, as well as presenting our MVP (Minimum Viable Product) for our project. By using tools such as Zoom and Discord, we had effortless communication with our group and our client. We used Google Calendar to set deadlines for client meetings, as well as code meetings for the group.

The expected large learning curve of all the technical tools was a relatively small problem. Throughout our collegiate years at Virginia Tech, we had become familiar with different programming language frameworks. Overall, our group had a decent understanding of Python. This helped us understand how Jupyter Notebook works and how to better utilize the cells for printing out important feedback [4]. As a group, we had experience with Parquet files. We now have a better understanding of Parquet files and how to utilize them. BeautifulSoup is a very useful tool for web scraping with Python. We found that BeautifulSoup helps us with reading Parquet files and extracting needed data from them [2][5][7]. We solved the issues faced with the previous team's code of having to read in a Parquet file one at a time, by refactoring and changing the code to allow for multiple files to be read [2]. This helped with convenience and allowed for better performance overall.

D. Future Work

There are several areas for improvement. Our team has managed to extract internal and external links, raw text, meta tags content for description and keywords meta tags, colors, and images along with ad trackers. We hope that our codebase can further help with extracting other pieces of information from the Parquet files. We also hope our code base can easily output all the data we have extracted for different state Parquet files. However, the timing and performance of this could be further improved upon. Future teams should be able to take our codebase and apply that to extract information from other state's Parquet files in order to help with researching state tourism efforts [2]. This addition can allow research teams to see the variance of colors in all states through the stacked bar charts. Future teams should take a look at the Assessment of Implementation section, for optimization and refactoring concerns.

One key problem we noticed was overall performance of the application when running the two files for data extraction and visualization. For some state's Parquet files, it could take anywhere from one to almost four hours. Future teams and research could help improve the performance of our two programs. An applicable solution to speedup performance may be to use different sorting and recursion algorithms for the application. There are also ways to go through third-party applications and/or a different API than Python that may help with reading through directories and files on a computer [2][7].

XI. Acknowledgements

Florian Zach, Ph.D. - Assistant Professor in the Howard Feiertag Department of Hospitality and Tourism Management. Email: florian@vt.edu

Xinyue (Cyrus) Wang - Ph.D. Student in the Department of Computer Science. Email: xw0078@vt.edu

NSF IIS-1619028, Global Event and Trend Archive Research (GETAR)

NSF CMMI-1638207, Coordinated, Behaviorally-Aware Recovery for Transportation and Power Disruptions (CBAR-tpd)

US State Tourism Spring 2020 Team, <http://hdl.handle.net/10919/98257>

US State Tourism Spring 2019 Team, <http://hdl.handle.net/10919/92622>

XII. References

- [1] Doan, Viet, et al. "Tourism Destination Websites." VTechWorks, Virginia Tech, 8 May 2019, <http://hdl.handle.net/10919/92622>, accessed 4/23/2021.
- [2] "Apache Parquet.", Apache Software Foundation, 2018, Parquet.apache.org/, accessed 4/23/2021.
- [3] "Pandas - Python Data Analysis Library." Pandas, NumFOCUS, 2020, pandas.pydata.org/, accessed 4/23/2021.
- [4] "Project Jupyter." 2020, jupyter.org/, accessed 4/23/2021.
- [5] Richardson, Leonard. "Beautiful Soup Documentation." Beautiful Soup 4.9.0 Documentation, 2020, www.crummy.com/software/BeautifulSoup/bs4/doc/, accessed 4/23/2021.
- [6] "The World's Most Popular Data Science Platform." Anaconda, 2020, www.anaconda.com/, accessed 4/23/2021.
- [7] Van Rossum, G. & Drake, F.L., 2009. *Python 3 Reference Manual*, Scotts Valley, CA, <https://docs.python.org/3/reference/>, accessed 4/23/2021.
- [8] Howall, M., Homebrew., 2021, <https://brew.sh/>, accessed 4/23/2021.
- [9] Cloud, P., *pyarrow*. PyPI., 2021, <https://pypi.org/project/pyarrow/>, accessed 4/23/2021.
- [10] Heikkinen, I., *HTML*. HTML Standard. 2021, <https://html.spec.whatwg.org/>, accessed 4/23/2021.
- [11] J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007. accessed 4/23/2021.
- [12] Pezoa, F. et al., 2016. Foundations of JSON schema. In Proceedings of the 25th International Conference on World Wide Web. pp. 263–273. accessed 4/23/2021.