Assignment 1 - Documentation
INFO 7390
Advanced Data Science and Architecture

Working with Edgar datasets: Wrangling, Pre-processing and Exploratory Data Analysis

Presented by:

DhanishaDamodarPhadate
001859234
phadate.d@husky.neu.edu

Palak Sharma
001834478
sharma.pala@husky.neu.edu

Dharani Thirumalaisamy
001887922
thirumalaisamy.d@husky.neu.edu

CONTENT

1. Overview

- Programming Language and Libraries 3
- Logging 4

2. Abstract

- Problem 1 6
- Problem 2 6

3. Approach

- Problem 1- Data wrangling Edgar data from text files

  a) Part 1: Parse files

  - Methodology 7
  - Process overview ..... 7 - 14

  b)Part 2: Dockerize this pipeline

  - Docker Steps to build and run the code 15
  - Giving Inputs 16
  - Amazon s3 bucket 16
  - Handling Exceptions 16

- Problem 2 - Data Analysis
  a) Methodology 17
  b) Process Overview 17- 31
  - Handling missing data 18
  - Summary Metrics ..... 20 -25
    - Statistical Method
    - Graphical Method
  - Anomalies detection 24
  - Analysis on year 2008 data set 25

Conclusion
References

OVERVIEW :

Programming Language and Libraries

We have used Python programming language and following libraries to solve this assignment.

Libraries used:

- **urllib** - allows access to websites via program
- **BeautifulSoup(bs4)** - web scraping
- **OS** - operating system dependant functionality
- **sys** - access to variables used by the interpreter
- **zipfile** - used for zipping files
- **csv** - write to .csv format
- **logging** - track events
- **boto** - python package that provides interface to Amazon Web Services
- **boto.s3** - creates connection to interact with the server
- **shutil** – to remove files and directory
- **Glob** – to read files efficiently
- **Pandas** – put data from csv files into data frames
- **itertools**– creates iterators for efficient looping
- **argparse** – to get command line inputs and arguments
- **numpy**– for computing on data
- **statistics** – to obtain functions like mode
- **matplotlib**– 2D plotting library for visualization

Tools used: Jupyter Notebook, Amazon S3 bucket, Docker, Github, Sublime Command line.

Logging:

We have used the logging module in python which will log the operations with the timestamp, levelname, file name, line number and message that we have customized.

There are 5 levels in logging module:

Debug: Detailed information, typically of the interest only when diagnosing problems

Info: Confirmation that things are working as expected

Warning: An indication that something unexpected happenedor indicative of some problem in the near future

Error: Due to more problem, the software has not been able to perform some function

Critical: A serious error, indicating that the program itself may be unable to continue running

We have used 3 levels- Info, error and warning in our logs.

Below two handlers are used to log the output.

File handler – This willlog all the level messagesinto the log file.

Stream handler – This will log the level messages into console.

Logging code snippet:

```
#create logger
logger = logging.getLogger()
logger.setLevel(logging.DEBUG) #setting level to debug

#creates file handler which logs all the level messages
fh = logging.FileHandler('Log_file2.log' , mode = 'w')
fh.setLevel(logging.DEBUG) #setting level to handler to debug

formatter = logging.Formatter('[%(asctime)s] %(levelname)8s --- %(message)s' +
                              '(%(filename)s:%(lineno)s)' ,datefmt = '%Y-%m-%d %H:%M:%S')

if (logger.hasHandlers()):
    logger.handlers.clear()

#if handler_console is none :
handler_console = logging.StreamHandler(stream = sys.stdout)

#then add it back
handler_console.setFormatter(formatter)
handler_console.setLevel(logging.INFO)
logger.addHandler(handler_console)

fh.setFormatter(formatter)
logger.addHandler(fh)
```

```
#create logger
logger = logging.getLogger()
logger.setLevel(logging.DEBUG) #setting level to debug

#creates file handler which logs all the level messages
fh = logging.FileHandler('Log_file1.log' , mode = 'w')
fh.setLevel(logging.DEBUG) #setting level to handler to debug

formatter = logging.Formatter('[%(asctime)s] %(levelname)8s --- %(message)s' +
                              '(%(filename)s:%(lineno)s)' ,datefmt = '%Y-%m-%d %H:%M:%S')

if (logger.hasHandlers()):
    logger.handlers.clear()

#if handler_console is none :
handler_console = logging.StreamHandler(stream = sys.stdout)

#then add it back
handler_console.setFormatter(formatter)
handler_console.setLevel(logging.INFO)
logger.addHandler(handler_console)

fh.setFormatter(formatter)
logger.addHandler(fh)
```

ABSTRACT

PROBLEM 1:

In this part of the question, we have automated the generation of URL based on the CIK and Accession number. Once the URL is formed, all the 10-q links will be parsed and tables from the forms will be scraped and stored as individual tables which we have later zipped and pushed it to S3 bucket. The whole process is automated using Docker as the last step of this problem.

PROBLEM 2:

This part deals with the extraction of the log files present on the link given below:

https://www.sec.gov/dera/data/edgar-log-file-data-set.html

and then after downloading the zipped files inside a directory, these files are unzipped to access the CSVs and the data inside them. A series of pre-processing and data wrangling is applied on the 1st day(next one if the previous is not available) of each month of a particular year.The data is converted to a data frame and missing data is handled first , then the summary metrics in both statistical and graphical form is calculated. The analysis ends with calculating the anomalies in the data, if any.

APPROACH

Problem 1 Data wrangling Edgar data from text files:

Part 1: Parse files

Methodology:

1. Using the cik and accession number of a company, the url which contains the list of form is generated.
2. From that url, the url with 10-q forms are accessed.
3. From the 10-q form all the tables are searched.
    a) Searched for statistical data which has '$' and '%' in it.
4. Once the tables are separated from the other content, the data in each table is extracted.
    a) The data is present inside the <font> tag which is present inside <p>tag .
    b) To go inside the <p> tag we are first searching for all <p> tag in <td> which is present in <tr>.
    c) So once <font> tag is searched , the data is extracted in the form of text.
    d) There is a chance for the data not to be clean , so we are removing all the unwanted content.
5. After extracting all the data from the font , we are storing that in list form .
6. This is then converted to csv format where the rows are formed by the data from the <font> tag. All these csv files are stored in a separate folder which is generated automatically.
7. This folder which contains the csv files is then stored in a zip file.

Process overview:

Step 1:

Generating URL:

```
#automating url

url = ("https://www.sec.gov/Archives/edgar/data/")

new_cik = cik.lstrip('0')
new_accession_number = accession_number.replace("-","")
new_url = (url + new_cik + "/" + new_accession_number + "/" + accession_number + "-" + "index.html")
       # new_url.get_method = lambda: 'HEAD'
try :
    urllib.request.urlopen(new_url)
    logger.info("Url request is successful")
except urllib.request.HTTPError:
    logger.exception("HTTP url not found")
    exit()

new_url_1 = urllib.request.urlopen(new_url).read()
new_url_2 = bs(new_url_1 , 'html.parser')
```

1. The standard URL part for all the EDGAR data irrespective of their cik and accession number is stored in a variable called 'url'.

2. cik is generally a 10-digit number with '0's' preceding it. To generate the proper url we need to strip out the unwanted digits. So using .lstrip() function all the 0's are stripped.

3. To create the proper url, we need to omit all the '-' in accession number which is done using replace() function. This replaces ever '-' with "".

4. The newly created URL is stored in a variable called 'new_url'.

5. If the 'new_url' exist the log will output "Url request is successful" or else it will log "HTTP url not found" and exit the loop.
6. If the 'new_url' works, then the url is been read and that is parsed. This is stored in another variable called "new_url_2".

```
cik='0000051143'
accession_number = '0000051143-13-000007'

url = ("https://www.sec.gov/Archives/edgar/data/")

new_cik = cik.lstrip('0')
new_accession_number = accession_number.replace("-","")
new_url = (url + new_cik + "/" + new_accession_number + "/" + accession_number + "-" + "index.html
      # new_url.get_method = lambda: 'HEAD'
try :
    urllib.request.urlopen(new_url)
    logger.info("Url request is successful")
except urllib.request.HTTPError:
    logger.exception("HTTP url not found")
    exit()

new_url_1 = urllib.request.urlopen(new_url).read()
new_url_2 = bs(new_url_1 , 'html.parser')
```

```
[2018-02-14 21:48:08]     INFO --- Url request is successful(<ipython-input-1-aa50bba93e52>:50)
```

This is the output with proper cik and accession number.

Step 2:

Finding the 10-q form:

1. From the "new_url_2" which holds the parsed html page, all the <a> tags are found and that is stored in a variable called "containers".
2. In that containers , all the "href" labels are searched and the one with "10-q.htm" will be stored in "final_url".
3. The "final_url" is then parsed and stored in "scrape".

```
#finding the form-10q
#finding the form-10q
tbl = new_url_2.findAll('table',{'summary':'Document Format Files'})
tbll = tbl[0] # cob=nverting list into table
trl= tbl[0].findAll('tr') #extracting all tr of the the table

i=0
url=""
for i in range(1,len(trl)):
    td_list= trl[i].findAll('td')
    for j in range(0, len(td_list)+1):
        if (j==3):
            if(td_list[j].text == "10-Q"):
                url= td_list[j-1].text
                break
```

```
new_url_1 = urllib.request.urlopen(new_url).read()
new_url_2 = bs(new_url_1 , 'html.parser')

containers = new_url_2.findAll('a')

for ref in containers :
    required = ref.get("href")
    if "10q.htm" in required :
        final_url = "https://www.sec.gov" + required
        scraping_page = urllib.request.urlopen(final_url).read()
        logger.info("10-q form request is successful")
        scrape = bs(scraping_page,"html.parser")
        print(final_url)
```

```
[2018-02-14 21:54:50]     INFO --- Url request is successful(<ipython-input-2-b5efc4448d88>:50)
[2018-02-14 21:54:56]     INFO --- 10-q form request is successful(<ipython-input-2-b5efc4448d88>:65)
https://www.sec.gov/Archives/edgar/data/51143/000005114313000007/ibm13q3_10q.htm
```

The final_url is printed for the given cik and accession number.

Step 3:

Finding Tables from the form and cleaning it:

The 10-q form has not only tables, but also other types of headers and footers. So in order to extract all the tables first we need to separate out the tables from the unwanted content.

```
#extracting all div yag
scrape_page = scrape.select('div table')
rawlist_tables=[]
for tables_1 in scrape_page:
    for rows in tables_1.find_all('tr'):
        number_tables=0
        for col in rows.findAll('td'):
            if('$' in col.get_text() or '%' in col.get_text()):
                rawlist_tables.append(tables_1)
                number_tables=1;
                break;
                if(number_tables==1):
                    break;
```

1. All the tables in a html page will be inside the "table" tag which will always be present inside the "div" tag. So after inspecting the page , using select() function 'div table' tag is being found. select() function helps in finding two tags simultaneously. All the content under 'div table' is stored inside 'scrape_page' variable.

Output of scrape_page is printed below.

```
<table border="0" cellpadding="0" cellspacing="0" style="border-collapse:collapse;width:100%;">
<tr style="page-break-inside:avoid;">
<td style="padding:0in .7pt 0in .7pt;" valign="top" width="44%">
<p align="center" style="margin:0in;margin-bottom:.0001pt;page-break-after:avoid;text-align:center;"><b><u><font face="Times Ne
w Roman,serif" style="font-size:10.0pt;">New York</font></u></b></p>
</td>
<td style="padding:0in .7pt 0in .7pt;" valign="bottom" width="3%">
<p style="margin:0in;margin-bottom:.0001pt;page-break-after:avoid;"><font face="Times New Roman,serif" style="font-size:10.0p
t;"> </font></p>
</td>
<td style="padding:0in .7pt 0in .7pt;" valign="top" width="53%">
<p align="center" style="margin:0in;margin-bottom:.0001pt;page-break-after:avoid;text-align:center;"><b><u><font face="Times Ne
w Roman,serif" style="font-size:10.0pt;">13-0871985</font></u></b></p>
</td>
</tr>
<tr style="page-break-inside:avoid;">
<td style="padding:0in .7pt 0in .7pt;" valign="top" width="44%">
<p align="center" style="margin:0in;margin-bottom:.0001pt;page-break-after:avoid;text-align:center;"><font face="Times New Roma
n,serif" style="font-size:10.0pt;">(State of incorporation)</font></p>
</td>
<td style="padding:0in .7pt 0in .7pt;" valign="bottom" width="3%">
```

2. Once that is filtered out , the tables inside this should be filtered out. For that , the only thing that differentiates the tables from other content is '$' and '%'. So that is found by iterating through <font> tag. This value is stored in the variable called "rawlist_tables". Now , the "rawlist_tables" holds the contents that is present inside the tables in that page.

Step 4:

Extracting data from the table and Cleansing it:

```
for tables_2 in rawlist_tables:
    final_list = []
    for rows_1 in tables_2.find_all('tr'):
        row=[]
        for columns_1 in rows_1.findAll('td'):
            para = columns_1.find_all('p')
            if len(para)>0:
                for para_1 in para:
                    new_para = para_1.get_text().replace("\n"," ")
                    new_para_1 = new_para.replace("\xa0","")
                    row.append(new_para_1)
            else :
                new_col=columns_1.get_text().replace("\n"," ")
                new_col_2=new_col.replace("\xa0","")
                row.append(new_col_2)

        final_list.append(row)
```

1. Once the tables are found , the next step is to scrape the data from them. For this , again <font > tag searched for the content of tables.

```
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;">(Dollars in millions except
  per share amounts)</font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;"> </font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;"> </font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;">2013</font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;"> </font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;"> </font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;">2012</font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;"> </font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;"> </font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;">2013</font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;"> </font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;"> </font>
<font color="black" face="Times New Roman,serif" style="font-size:8.0pt;">2012</font>
<font color="black" face="Times New Roman,serif" style="font-size:10.0pt;">Revenue:</font>
<font color="black" face="Times New Roman,serif" style="font-size:10.0pt;"> </font>
<font color="black" face="Times New Roman,serif" style="font-size:10.0pt;"> </font>
<font color="black" face="Times New Roman,serif" style="font-size:10.0pt;"> </font>
```

2. Sometimes ,<font> tag will not have any content . To omit all those cases the len(styles) condition is used.

3. The content inside the <font> tag will hold other unwanted characters such as "\n" , "\xao" which has to be replaced. So function replace() is used to do that.

4. Once the table is ready , all the table data is appended to final_list.

```
'', '24,747', '', '', '72,052', '', '', '75,203'], ['Cost:', '', '', '', '', '', '', '', '', '', '', ''
'', '', '9,098', '', '', '9,515', '', '', '27,950', '', '', '29,285']]
['', '', '', '', '', '', '', '', '', '', '', '', '', ''], ['', '', '', 'ThreeMonths Ended   September
ıths Ended   September 30,'], ['(Dollarsinmillionsexcept   pershareamounts)', '', '', '2013', '', '', '
'', '', '2012'], ['Revenue:', '', '', '', '', '', '', '', '', '', '', '', ''], ['Services', '', '$', '1
26', '', '$', '42,811', '', '$', '44,279'], ['Sales', '', '', '8,987', '', '', '9,642', '', '', '27,735
''Financing', '', '', '509', ' ', '', '479', '', '', '1,506', ' ', '', '1,500'], ['Total   revenue', ''
'', '24,747', '', '', '72,052', '', '', '75,203'], ['Cost:', '', '', '', '', '', '', '', '', '', '', ''
'', '', '9,098', '', '', '9,515', '', '', '27,950', '', '', '29,285'], ['Sales', '', '', '2,975', '', '
'9,108', '', '', '10,003']]
['', '', '', '', '', '', '', '', '', '', '', '', '', ''], ['', '', '', 'ThreeMonths Ended   September
ıths Ended   September 30,'], ['(Dollarsinmillionsexcept   pershareamounts)', '', '', '2013', '', '', '
'', '', '2012'], ['Revenue:', '', '', '', '', '', '', '', '', '', '', '', ''], ['Services', '', '$', '1
26', '', '$', '42,811', '', '$', '44,279'], ['Sales', '', '', '8,987', '', '', '9,642', '', '', '27,735
''Financing', '', '', '509', ' ', '', '479', '', '', '1,506', ' ', '', '1,500'], ['Total   revenue', ''
'', '24,747', '', '', '72,052', '', '', '75,203'], ['Cost:', '', '', '', '', '', '', '', '', '', '', ''
```

The data that is appended to the final_list

Step 5:

Saving it as .csv files:

```python
with open(os.path.join('FINAL_TABLES' ,  'DATA' + str(rawlist_tables.index(tables_2)) + '.csv'), 'w') as csv_file:
    csv_writer = csv.writer(csv_file)
    csv_writer.writerows(final_list)
    logger.info("Table has been extracted in csv format")
```

1. Directory 'FINAL_TABLES' have been formed using "os.mkdir()" function.

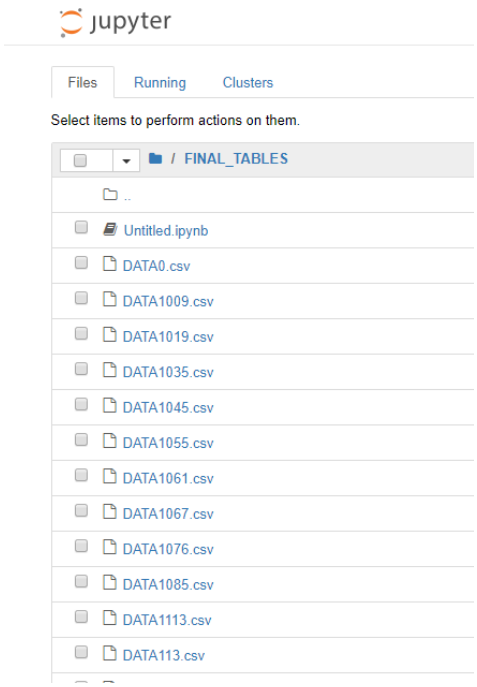
```python
tracting the tables
    if not os.path.exists('FINAL_TABLES'):
        try:
            os.makedirs('FINAL_TABLES')
        except OSError as exc: # Guard against race condition
                logger.error("OSError" + str(exc))
```

2. The tables are iterated by indexing "tables_2".

3. All the files are written as csv files using "csv.writer()" function.

```
[2018-02-14 22:50:13]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:13]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:13]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:13]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:13]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:13]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:13]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
[2018-02-14 22:50:14]     INFO --- Table has been extracted in csv format(<ipython-input-4-54ef8afc0651>:150)
```

The log output



All the csv files are created and stored in FINAL_TABLES folder.

Desired output :

	ThreeMonths Ended	Septemb	Nine Months Ended	September 30,	
(Dollarsinmillionsexcept pers	2013	2012	2013		2012
Revenue:					
Services	$ 14,225	$ 14,626	$ 42,811	$	44,279
Sales	8,987	9,642	27,735		29,424
Financing	509	479	1,506		1,500
Total revenue	23,720	24,747	72,052		75,203
Cost:					
Services	9,098	9,515	27,950		29,285
Sales	2,975	3,242	9,108		10,003
Financing	268	258	805		784
Total cost	12,341	13,016	37,863		40,072

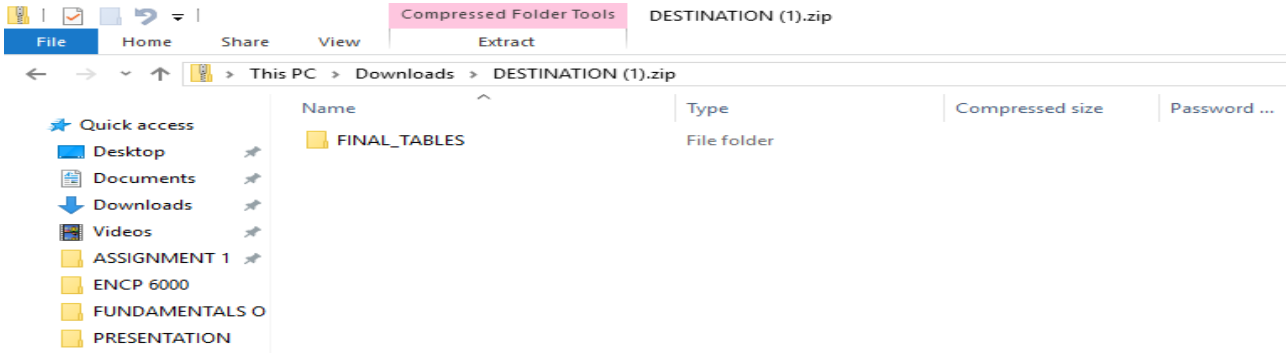
Step 6:

Zipping all the files:

```
# zipping all the csv files

zip_file = zipfile.ZipFile('DESTINATION.zip' , 'w',zipfile.ZIP_DEFLATED)
for tables_2 in rawlist_tables:
    zip_file.write(os.path.join('FINAL_TABLES' ,  'DATA' + str(rawlist_tables.index(tables_2)) + '.csv'))
    logger.info("All files are zipped")
    zip_file.close()
```

1. zipfile() function is used here. Using that "DESTINATION.zip" file is created and all the files in the FINAL_TABLES are zipped in here.



Part 2: Dockerize this pipeline

Docker Steps to build and run the code:

To build and run the image locally

1. create a docker file using below line

    vim mydockerfile

2. Build the image

    docker build -f mydockerfile -t assignpart1

3. Run the image to create container

    docker run -e CIK=cik -e Accession_number=accno -e

Access_key=acckey -e Secret_key=seckey -e Input_location=iloc -tiassignpart

4. Tag the image

docker tag <image_id>dhanisha/assignpart1:latest

5. Push the docker image to docker hub

docker push dhanisha/assignpart1

6. To pull the docker image from docker hub

docker pull dhanisha/assignpart1

7. Run the image pulled from docker hub using following commands

docker run -e CIk=cik -e Accession_number=accno -e Access_key=acckey -e Secret_key=seckey -e Input_location=iloc -ti dhanisha/assignpart1

Giving Inputs:

a) Here we are using arg parser to pass arguments that the interpreter needs.

b) It takes cik, accession number, accesskey , secret key and then the location(to create bucket) in the same order.

c) After it takes the input, it takes the length of the parameter.

Using Amazon S3 bucket:

Amazon Simple Storage Service (Amazon S3) is a web service that provides highly scalable cloud storage. Amazon S3 provides easy to use object storage, with a simple web service interface to store and get any amount of data from anywhere on the web.

Use library Boto s3 to access Amazon Simple Storage Service (Amazon S3) using AWS Python SDK.

Follow below steps to upload files to amazon S3

1. Create S3 client passing access key, secret key as a parameter
2. Create a bucket giving it a name and passing location as a parameter
3. Upload the file to bucket specifying bucket name and file name.

```
#uploading files to amazon s3
'''BucketAlreadyOwnedByYou errors will only be returned outside of the US Standard region.
Inside the US Standard region (i.e. when you don't specify a location constraint), attempting to recreate a bucket you a
logger.info("Uploading files to amazon")
try:

    buck_name="ads-part1-assignment"

    S3_client = boto3.client('s3', Input_location, aws_access_key_id= Access_key, aws_secret_access_key= Secret_key)

    if Input_location == 'us-east-1':
            S3_client.create_bucket(
                Bucket=buck_name
            )
    else:
        S3_client.create_bucket(
                Bucket=buck_name,
                CreateBucketConfiguration={'LocationConstraint': Input_location},
            )
    logging.info("connection successful")
    S3_client.upload_file("Extracted_tables.zip", buck_name, "Extracted_tables.zip")
    S3_client.upload_file("Log_file1.log", buck_name, "Log_file1.log")
    logging.info("Files uploaded successfully")
except Exception as e:
    logging.error("Error uploading files to Amazon s3" + str(e))

# In[ ]:
```

Handling exceptions:

1. Validation of CIK and Accession number:
   If the CIK and Accession number given is incorrect , it logs Invalid CIK and accession number and exits the code . Moreover, it throws HTTP Error as the url won't be formed.

2. Validating Amazon access key and Secret key :
   When proper credentials are given , it connects to the HTTP and forms a s3 bucket in aws or else it throws a HTTP error and exits the program.

Problem 2Missing Data Analysis:

Methodology:

1. The user is asked to input a year, awsaccessKey ,awssecretKey and the location.
   On the basis of the year provided, an url will be generated, which will help the user to access the log files from EDGAR site of the entered year.
2. All the scenarios are taken into consideration to check if all the four inputs are valid or not. If even one of them is wrong, an error is thrown.

3. Apart from the validations on the inputs, during the file extractions, if the log file is not available for any month or for any day , then the next available files are tried, if none of the data is available then the exception is thrown.
4. The data extracted is cleaned via missing data calculation and then summary metrics is calculated for the same followed by visualizations.
5. Finally, the results are stored in one directory and uploaded to AWS 3.

Process overview:

Step 1:

Handle Missing values:

a. Reading the csv file in to a panda data frame and Finding out the null values in each column of the dataframe using :
#data.isnull().sum()

b. Working on different columns with null values:

b.1 Browser: Using the "MAXIMUM LIKELIHOOD" method and replacing the NaN values with the maximum used browser for that day of the month of the given year using the below commands:
#**d =df.apply(pd.value_counts)**
#data['browser'].replace(np.nan,d.index[0], inplace = True)

b.2Size (NUMERICAL DATA) : The NaN values of the size column are handled via "IMPUTATION" method. The values of file size are grouped on the basis ofextention and then the NaN values are replaced by the mean of the size. The same is performed on 4 different set of extentions (.xml,.htm.,.txt,.).
#s=data[['extention','size']].groupby(data['extention'].str.contains('txt'))['size'].mean().reset_index(name='mean').sort_values(['mean'],ascending=False)
#data.loc[(data['size'].isnull()) & (data['extention'].str.contains('txt'))]
#data.loc[(data['size'].isnull()) & (data['extention'].str.contains('txt'))] = s
#data.loc[(data['size'].isnull()) & (data['extention'].str.contains('txt'))]

b.3 For the CATEGORICAL DATA columns (cik,accession,ip,date,time) : Since , these are the basis of all the predictions and conclusions, therefore dropping all the rows which have NaN values in these coloumns using the below commands.

#data.dropna(subset=['cik'],inplace=True)
#data.dropna(subset=['accession'],inplace=True)
#data.dropna(subset=['ip'],inplace=True)
#data.dropna(subset=['date'],inplace=True)
#data.dropna(subset=['time'],inplace=True)

b.4For the categorical columns (code,zone,extention,idx,find) : These columns are used for calculating the summary metrics , therefore dropping the NaN value rows
will make a problem in the analysis, therefore performing imputation and filling the values with the max value of the corresponding column.

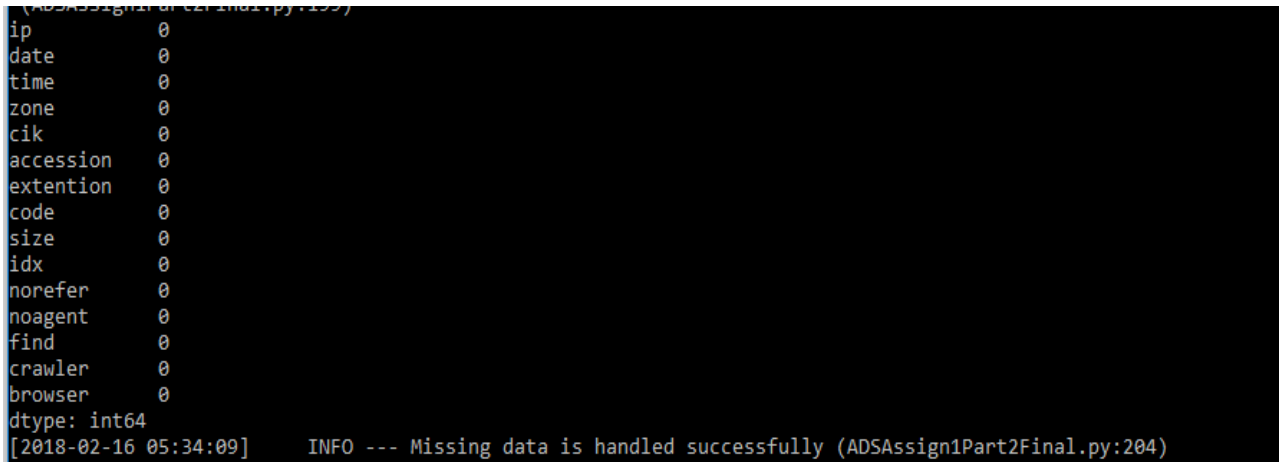
#data['code'].fillna(data['code'].max(),inplace=True)
#data['zone'].fillna(data['zone'].max(),inplace=True)
#data['extention'].fillna(data['extention'].max(),inplace=True)
#data['idx'].fillna(data['idx'].max(),inplace=True)
#data['find'].fillna(data['find'].max(),inplace=True)

b.5For all the remaining columns :

#data['norefer'].fillna(1,inplace=True)
#data['noagent'].fillna(1,inplace=True)
#data['crawler'].fillna(0,inplace=True)

As soon as the different functions are performed, they are getting logged in the logger file, indicating every steps for missing data handling.

After all the steps are built successfully, the value of the NaN values throughout the data is calculated again and the expected output is:



----------Missing data handling is achieved----------

Step 2 :

Summary Metrics:

a. The cleaned data is now worked upon for various summary metrics both in statistical and graphical form. The first command used is the **describe()** command to get a brief view of the data statistics.

b. **STATICTICAL METHOD:**
b.1To calculate which class of IP addresses(Class A, Class B , Class C) has done the maximum EDGAR fillings using mode :

The following analysis helps us in finding out the group of companies which file the maximum of Q10 forms, or who are searched the most , which will give us a
Prediction parameter that which companies are likely to follow the trend, this is done on the basis of IP classes. Class A companies are the big companies likes Google and Yahoo, and reduces with Class B and Class C.

```
ClassAlist = []
ClassBlist = []
Others = []
for i in range(0,len(df.index)):
    a = df.iloc[i]
    octects = a.str.split('.',expand = True)
    o = octects.iloc[0,0]
    if 0<int(o)<127:
        ClassAlist.append(int(o))
    elif 128<int(o)<182:
        ClassBlist.append(int(o))
    else:
        Others.append(int(o))
if len(ClassAlist) > (len(ClassBlist) | len(Others)):
    logger.info("The Companies who have filled maximum are the ones having ClassA ips ")
if len(ClassBlist) > (len(ClassAlist) | len(Others)):
    logger.info("The Companies who have filled maximum are the ones having ClassB ips ")
if len(Others) > (len(ClassBlist) | len(ClassAlist)):
    logger.info("The Companies who have filled maximum are the ones having ClassC ips ")
```

b.2Mean and Median sizes for each Browser :

The following summary metric helps in understanding what browsers are the most used browsers, or where the trend is moving towards like which browsers are about to be closed.

```
logger.info("Mean and Median sizes for each Browser")
brow_df = data.groupby('browser').agg({'size':['mean', 'median'],'crawler': len})
brow_df.columns = ['_'.join(col) for col in brow_df.columns]
data.reset_index(drop=True)
print(brow_df)
```

b.3Distinct count of ip per month i.e. per log file and status code on the basis ofip:

```
    data.reset_index(drop=True)

#Compute distinct count of ip per month i.e. per log file
    ipcount_df = df['ip'].nunique()
    logger.info("Compute distinct count of ip per month i.e. per log file")
    print(ipcount_df)

#Computing the count of status code on the basis of ip
    StCo_count=data[['code','ip']].groupby(['code'])['ip'].count().reset_index(name='count')
    logger.info("Computing the count of status code on the basis of ip")
    print(StCo_count)
    data.reset_index(drop=True)
```

b.4To find out the 15 top searched CIKs :

The following summary metric is yet another prediction parameter, about the future scope of the top 15 maximum searched companies on the basis of their CIKs

```
#To find out the 15 top searched CIKs
    cik_df = pd.DataFrame(data, columns = ['cik'])
    d = cik_df.apply(pd.value_counts)
    logger.info("Top 15 most searched CIKs with the count")
    d.head(15)
    data.reset_index(drop=True)
```

b.5Everything on per day basis

Some of the values are measured on per day basis:

```
#1. Average of Size
Avg_size=data[['date','size']].groupby(['date'])['size'].mean().reset_index(name='mean')
logger.info("Average of file size is computed")
print(Avg_size)
#2. Number Of Requests
Req_day=data[['date','ip']].groupby(['date'])['ip'].count().reset_index(name='count')
logger.info("Number of request per day is computed")
print(Req_day)
```

----------Summary Metrics(Statistical) is Calculated Successfully----------

c. GRAPHICAL METHOD:

The same set of metrics is visualized using graphs :

```
#graph of no of status codes by browser
    try:
        logger.info("graphical analysis started")
        Num_of_codes=data[['browser','code']].groupby(['browser'])['code'].count().reset_index(name = 'count_code').sort_values
(['count_code'],ascending=False)
        data.reset_index(drop = True)
        print(Num_of_codes)
        x= np.array(range(len(Num_of_codes)))
        y= Num_of_codes['count_code']
        xticks1 = Num_of_codes['browser']
        plt.xticks(x,xticks1)
        plt.bar(x,y)
        plt.title('Count of status code for all the browsers')
        plt.ylabel('Count of codes')
        plt.xlabel('Browsers')
        plt.savefig('Graphical_images/countsperbrowser'+ str(i) +'.png',dpi=100)
        #plt.savefig(os.path.join('Graphical_images',str(val),'countsperbrowser.png'),dpi=100)
        #plt.show()
        plt.clf()
        logger.info("graphical analysis end")
    except Exception as e:
        logger.error(str(e))
        logging.error("Error plotting the graph ")
```

```
#graph for max cik(10) by IP used
    try:
        logger.info("graphical analysis started")
        Num_of_CIKs=data[['cik','ip']].groupby(['cik'])['ip'].count().reset_index(name='count').sort_values(['count'],ascending=False).head
(10)
        data.reset_index(drop=True)
        print(Num_of_CIKs)
        x = np.array(range(len(Num_of_CIKs)))
        y = Num_of_CIKs['count']
        xticks2 = Num_of_CIKs['cik']
        plt.xticks(x, xticks2)
        plt.bar(x,y)
        plt.title('Top 10 CIKs by IPs')
        plt.ylabel('Count of IPs')
        plt.xlabel('CIK-')
        #plt.savefig(os.path.join('Graphical_images',str(val),'CIKsbyIPcount.png'),dpi=100)
        plt.savefig('Graphical_images/CIKsbyIPcount'+ str(i) +'.png',dpi=100)
        #plt.show()
        plt.clf()
        logger.info("graphical analysis end")
    except Exception as e:
        logger.error(str(e))
        logging.error("Error plotting the graph ")
```

```
        logging.error("Error plotting the graph ")

#Graph of Mean of file size on the basis of code status

    try:
        Mean_size=data[['code','size']].groupby(['code'])['size'].mean().reset_index(name='mean').sort_values(['mean'],ascending=False)
        data.reset_index(drop=True)
        print(Mean_size)
        x = np.array(range(len(Mean_size)))
        y = Mean_size['mean']
        xticks3 = Mean_size['code']
        plt.xticks(x, xticks3)
        plt.bar(x,y)
        plt.title('filesize by codes')
        plt.ylabel('mean size')
        plt.xlabel('Code')
        #plt.savefig(os.path.join('Graphical_images',str(val),'MeanSizeByCode.png'),dpi=100)
        plt.savefig('Graphical_images/MeanSizeByCode'+str(i) +'.png',dpi=100)
        #plt.show()
        plt.clf()
    except Exception as e:
        logger.error(str(e))
        logging.error("Error plotting the graph ")
```

The output for the graphical summary metrics are given below for the particular year : 2018

```
#Graph for average file size by extension
    try:
        Avg_size=data[['extention','size']].groupby(['extention'])['size'].mean().reset_index(name='mean').sort_values
(['mean'],ascending=False).head(20)
        data.reset_index(drop=True)
        print(Avg_size)
        x = np.array(range(len(Avg_size)))
        y = Avg_size['mean']
        xticks4 = Avg_size['extention']
        plt.xticks(x, xticks4)
        plt.bar(x,y)
        plt.title('Avg File size by extention')
        plt.ylabel('MeanFileSize')
        plt.xlabel('Extention')
        #plt.savefig(os.path.join('Graphical_images',str(val),'filesizebyextention.png'),dpi=100)
        plt.savefig('Graphical_images/filesizebyextention'+str(i) +'.png',dpi=100)
        #plt.show()
        plt.clf()
    except Exception as e:
        logger.error(str(e))
        logging.error("Error plotting the graph ")
"                                                                                                                    "
```

Step 3:

Checking Anomalies :

Graphical approach is used for calculating the Anomalies.

Anomaly detection is performed on the size column. Since the column used is a single data frame, boxplot is used. However, a better visualization is possible through Scatter, and bar plot. Only if size is visualized with respect to some other variable.

```
#Anomalies in FileSize
    try:
        logger.info("Anomalies analysis started")
        data.boxplot(column='size',vert=True,sym='',whis=10,showfliers=False)
        plt.xticks(rotation=70)
        plt.title('Anomalies displayed on the file size')
        plt.ylabel('size')
        #plt.savefig(os.path.join('Graphical_images',str(val),'Anomalies.png'),dpi=100
        plt.savefig('Graphical_images/'+'Anomalies'+ str(i) +'.png',dpi=100)
        #plt.show()
        logger.info("Anomalies analysis ended")
    except Exception as e:
        logger.error(str(e))
        logging.error("Error plotting the graph ")

    i = i+1
```

```
#uploading files to amazon s3
'''BucketAlreadyOwnedByYou errors will only be returned outside of the US Standard region.
Inside the US Standard region (i.e. when you don't specify a location constraint), attempting to recreate a bucket you already own will
succeed.'''
try:

    buck_name="ads-part2-assign"

    S3_client = boto3.client('s3', Location, aws_access_key_id= accessKey, aws_secret_access_key=secretKey)

    if Location == 'us-east-1':
            S3_client.create_bucket(
                Bucket=buck_name,
            )
    else:
        S3_client.create_bucket(
                Bucket=buck_name,
                CreateBucketConfiguration={'LocationConstraint': Location},
            )
    logging.info("Connection is successful")
    S3_client.upload_file("Graphical_images", buck_name, "Graphical_images")
    S3_client.upload_file("'Log_file2.log'", buck_name, "'Log_file2.log'")
    logging.info("Files uploaded successfully")
except Exception as e:
    logging.error("Error uploading files to Amazon s3" + str(e))
```

<u>EVERYTHING FOR YEAR 2008</u>:

ZIPPED and UNZIPPED Directories:

Two directories are created inside the docker container for storing all the zipped files we fetched using the url. And then all the zipped CSVs are then extracted inside the directory unzipped file.

```
[2018-02-16 10:17:04]     INFO --- Zipped file directory created!! (ADSAssign1Part2Final.py:87)
[2018-02-16 10:17:04]     INFO --- UnZipped file directory created!! (ADSAssign1Part2Final.py:95)
```

RETRIVING THE ZIPPED files from Internet

```
[2018-02-16 10:17:19]     INFO --- Retrieving zipped log file (ADSAssign1Part2Final.py:108)
[2018-02-16 10:17:25]     INFO --- Retrieving zipped log file (ADSAssign1Part2Final.py:108)
[2018-02-16 10:17:37]     INFO --- Retrieving zipped log file (ADSAssign1Part2Final.py:108)
[2018-02-16 10:17:52]     INFO --- Retrieving zipped log file (ADSAssign1Part2Final.py:108)
[2018-02-16 10:17:59]     INFO --- Retrieving zipped log file (ADSAssign1Part2Final.py:108)
```

EXTRACTING the CSVs month by month for the first day available:

```
[2018-02-16 10:19:06]     INFO --- log20080101.csv successfully extracted to folder: unzippedfiles. (ADSAssign1Part2Final.py:125)
[2018-02-16 10:19:07]     INFO --- log20080201.csv successfully extracted to folder: unzippedfiles. (ADSAssign1Part2Final.py:125)
[2018-02-16 10:19:07]     INFO --- log20080301.csv successfully extracted to folder: unzippedfiles. (ADSAssign1Part2Final.py:125)
[2018-02-16 10:19:08]     INFO --- log20080401.csv successfully extracted to folder: unzippedfiles. (ADSAssign1Part2Final.py:125)
[2018-02-16 10:19:10]     INFO --- log20080501.csv successfully extracted to folder: unzippedfiles. (ADSAssign1Part2Final.py:125)
[2018-02-16 10:19:11]     INFO --- log20080601.csv successfully extracted to folder: unzippedfiles. (ADSAssign1Part2Final.py:125)
[2018-02-16 10:19:13]     INFO --- log20080701.csv successfully extracted to folder: unzippedfiles. (ADSAssign1Part2Final.py:125)
```

GRAPHICAL_IMAGES directory is created to store all the png files at one place :

```
[2018-02-16 10:19:22]     INFO --- log20081201.csv successfully extracted to folder: unzippedfiles. (ADSAssign1Part2Final.py:125)
[2018-02-16 10:19:22]     INFO --- Graphical_images directory created!! (ADSAssign1Part2Final.py:144)
```

MISSING DATA ANALYSIS

a. Started with the calculation of which columns have missing value and how many are missing.

```
[2018-02-16 10:19:24]     INFO --- Calculated the missing values in each coloumn (ADSAssign1Part2Final.py:148)
ip                0
date              0
time              0
zone              0
cik               0
accession         0
extention         0
code              0
size           3729
idx               0
norefer           0
noagent           0
find              0
crawler           0
browser      125368
```

b. All the handling functions explained above are performed on this file of 2018

```
[2018-02-16 10:19:06]     INFO --- log20080101.csv
```

```
[2018-02-16 10:19:24]    INFO --- Finding the NaN values in each coloumn (ADSAssign1Part2Final.py:150)
[2018-02-16 10:19:24]    INFO --- Working on the Browser Coloumn (ADSAssign1Part2Final.py:151)
[2018-02-16 10:19:24]    INFO --- Grouping the values of all the browsers and storing in a dataframe (ADSAssign1Part2Final.py:154)
[2018-02-16 10:19:24]    INFO --- Counting the frequency of each browser type used in descending order (ADSAssign1Part2Final.py:157)
[2018-02-16 10:19:24]    INFO --- Selecting the brower at the index 0(as it will be the maximum used browser) (ADSAssign1Part2Final.py:160)
[2018-02-16 10:19:24]    INFO --- confirming that no NaN values are present on the browser coloumn (ADSAssign1Part2Final.py:163)
[2018-02-16 10:19:24]    INFO --- Working on the Size Coloumn (ADSAssign1Part2Final.py:164)
[2018-02-16 10:19:24]    INFO --- Replacing the file size for ext : txt, by the mean of all the file size corresponding to txt (ADSAssign1Part2Final.py:165)
[2018-02-16 10:19:27]    INFO --- Replacing the file size with NaN values for ext : htm, by the mean of all the file size corresponding to htm (ADSAssign1Part2Final.
:171)
[2018-02-16 10:19:28]    INFO --- Replacing the file size with NaN values for ext : xml, by the mean of all the file size corresponding to xml (ADSAssign1Part2Final.
:177)
[2018-02-16 10:19:29]    INFO --- To check how many NaN values are remaining  (ADSAssign1Part2Final.py:183)
[2018-02-16 10:19:29]    INFO --- Replacing the file size for rest of the files with the mean of file size of txt extension, as it is the max used (ADSAssign1Part2Fi
l.py:184)
```

After performing this, repeating a again, we can see that all the missing values are being taken care of :

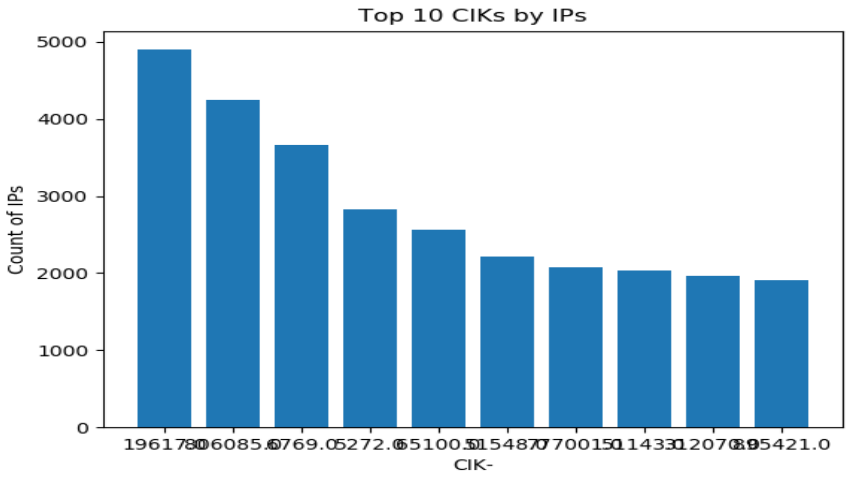
```
[2018-02-16 10:19:29]    INFO --- Working on all other coloumns (ADSAssign1Part2Final.py:187)
[2018-02-16 10:19:29]    INFO --- If cik,Accession,ip,date are empty fields drop the records (ADSAssign1Part2Final.py:188)
[2018-02-16 10:19:29]    INFO --- Calculating the max categorical value in other coloumns( code, zone,extention,idx,find) and filling the NaNs (ADSAssign1Part2Final.py
:194)
[2018-02-16 10:19:29]    INFO --- Filling empty values with Categorical Values for coloumns (norefer,noagent,nocrawler) (ADSAssign1Part2Final.py:201)
ip          0
date        0
time        0
zone        0
cik         0
accession   0
extention   0
code        0
size        0
idx         0
norefer     0
noagent     0
find        0
crawler     0
browser     0
dtype: int64
[2018-02-16 10:19:29]    INFO --- Missing data is handled successfully (ADSAssign1Part2Final.py:206)
```

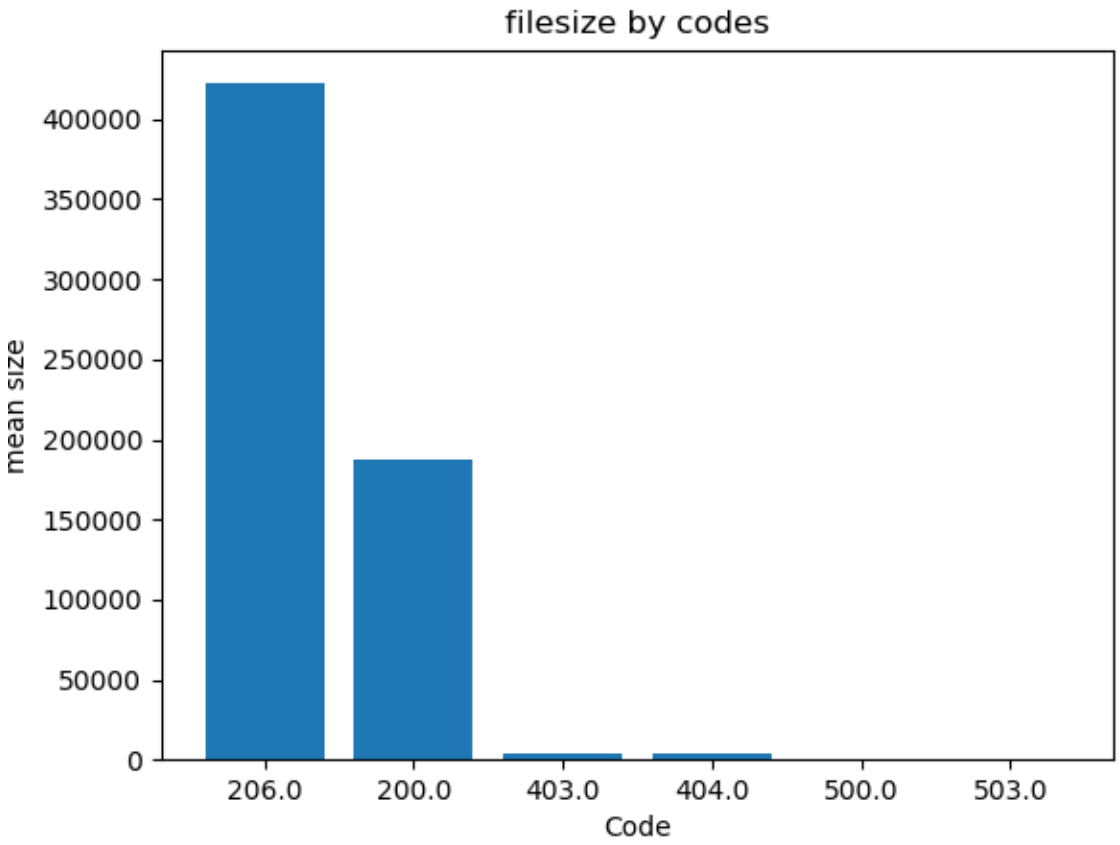
SUMMARY METRICS for the same

```
dtype: int64
[2018-02-16 10:19:29]    INFO --- Missing data is handled successfully (ADSAssign1Part2Final.py:206)
[2018-02-16 10:19:29]    INFO --- Calculating Summary metrics of clean data (ADSAssign1Part2Final.py:209)
[2018-02-16 10:19:30]    INFO --- To calculate which class of IP addresses(Class A, Class B , Class C) has done the maximum EDGAR fillings using mode (ADSAssign1Part2F
inal.py:212)
[2018-02-16 10:23:02]    INFO --- The Companies who have filled maximum are the ones having ClassA ips  (ADSAssign1Part2Final.py:228)
[2018-02-16 10:23:02]    INFO --- Mean and Median sizes for each Browser (ADSAssign1Part2Final.py:235)
          size_mean  size_median  crawler_len
browser
fox     1.155820e+05       9346.0         23.0
iem     3.717001e+06    3717001.0          1.0
iph     5.509095e+05     550909.5          2.0
lin     4.606733e+05      15420.5        256.0
mac     3.048609e+05       8109.0       1291.0
nie     3.722820e+05      15433.0        277.0
opr     4.817409e+03       3559.0         22.0
rim     1.322456e+06     312067.5         22.0
win     1.650593e+05       9207.5     159174.0
[2018-02-16 10:23:02]    INFO --- Top 15 most searched CIKs with the count (ADSAssign1Part2Final.py:244)
[2018-02-16 10:23:02]    INFO --- Compute distinct count of ip per month i.e. per log file (ADSAssign1Part2Final.py:250)
3895
[2018-02-16 10:23:02]    INFO --- Computing the count of status code on the basis of ip (ADSAssign1Part2Final.py:255)
    code   count
0  200.0  160021
1  206.0     956
2  404.0      91
[2018-02-16 10:23:02]    INFO --- Average of file size is computed (ADSAssign1Part2Final.py:262)
        date          mean
0  2008-01-01  167162.000143
[2018-02-16 10:23:02]    INFO --- Number of request per day is computed (ADSAssign1Part2Final.py:266)
        date   count
0  2008-01-01  161068
[2018-02-16 10:23:02]    INFO --- Mean of file size on the basis of code status (ADSAssign1Part2Final.py:270)
    code          mean
0  200.0  167240.206804
1  206.0  169654.670502
2  404.0    3451.000000
[2018-02-16 10:23:02]    INFO --- Summary metrics computed succesfully!! (ADSAssign1Part2Final.py:273)
```

GRAPHICAL REPRESENTATIONS

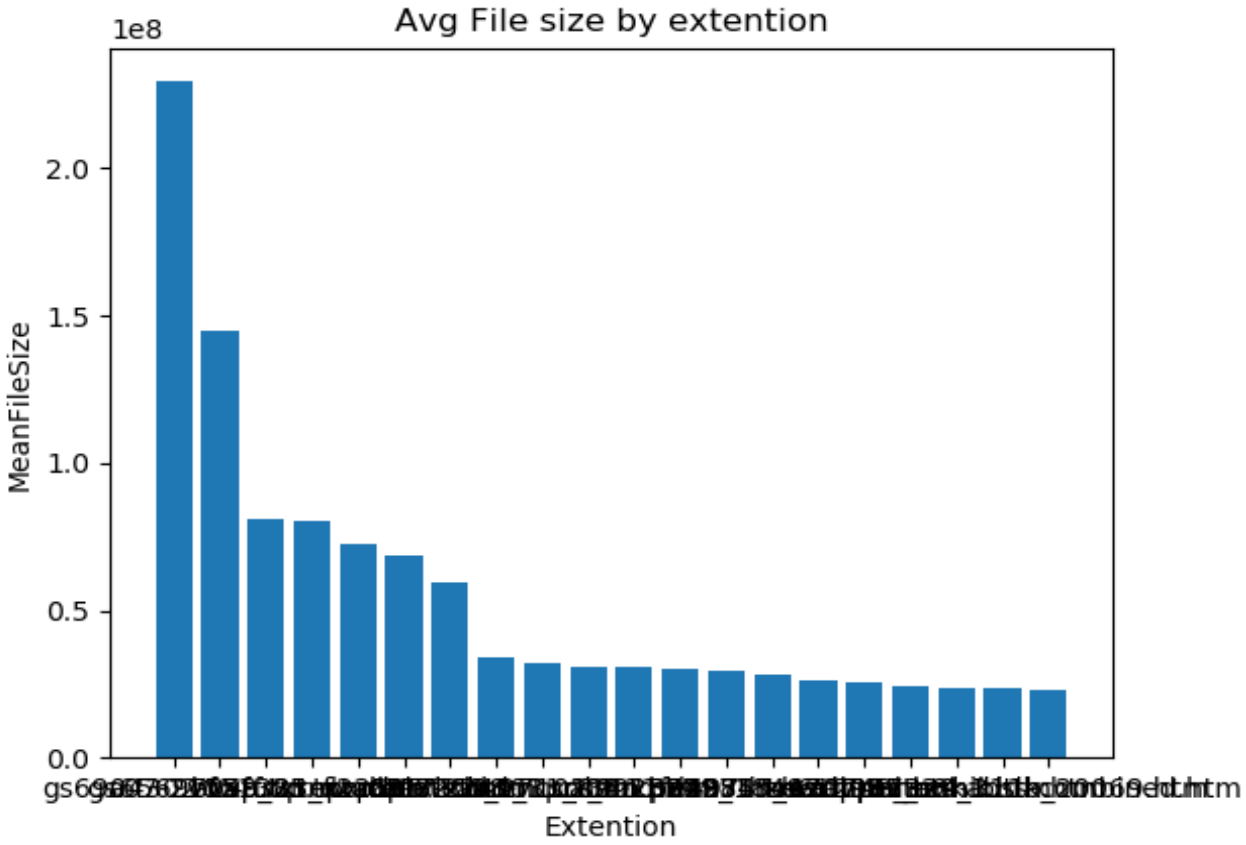
```
[2018-02-16 10:53:27]     INFO --- graphical analysis started (ADSAssign1Part2Final.py:280)
   browser  count_code
10     win      753160
4      mac        6378
3      lin        5492
5      mie        1699
0      fox         233
8      rim          52
6      opr          47
1      iem          10
2      iph           6
9      saf           2
7      oth           1
[2018-02-16 10:53:27]     INFO --- graphical analysis end (ADSAssign1Part2Final.py:296)
[2018-02-16 10:53:27]     INFO --- graphical analysis started (ADSAssign1Part2Final.py:303)
          cik  count
382    19617.0   4896
5844  806085.0   4242
141    6769.0    3666
104    5272.0    2823
1277   65100.0   2565
1050   51548.0   2209
4878  777001.0   2082
1040   51143.0   2027
2551  312070.0   1959
9771  895421.0   1908
[2018-02-16 10:53:27]     INFO --- graphical analysis end (ADSAssign1Part2Final.py:319)
    code           mean
1  206.0  422346.549425
0  200.0  187669.064187
2  403.0    3632.000000
3  404.0    3451.000000
4  500.0     525.000000
5  503.0     323.000000
                    extention          mean
35023          gs6904302-fwp.txt  2.294876e+08
35020          gs6569724-fwp.txt  1.445442e+08
2526         a07-21558_25npx.htm  8.108014e+07
50413            v051383_fwp.htm  8.008723e+07
15161      boafuturecomplete.htm  7.264694e+07
20683                  d22815.htm  6.887880e+07
```

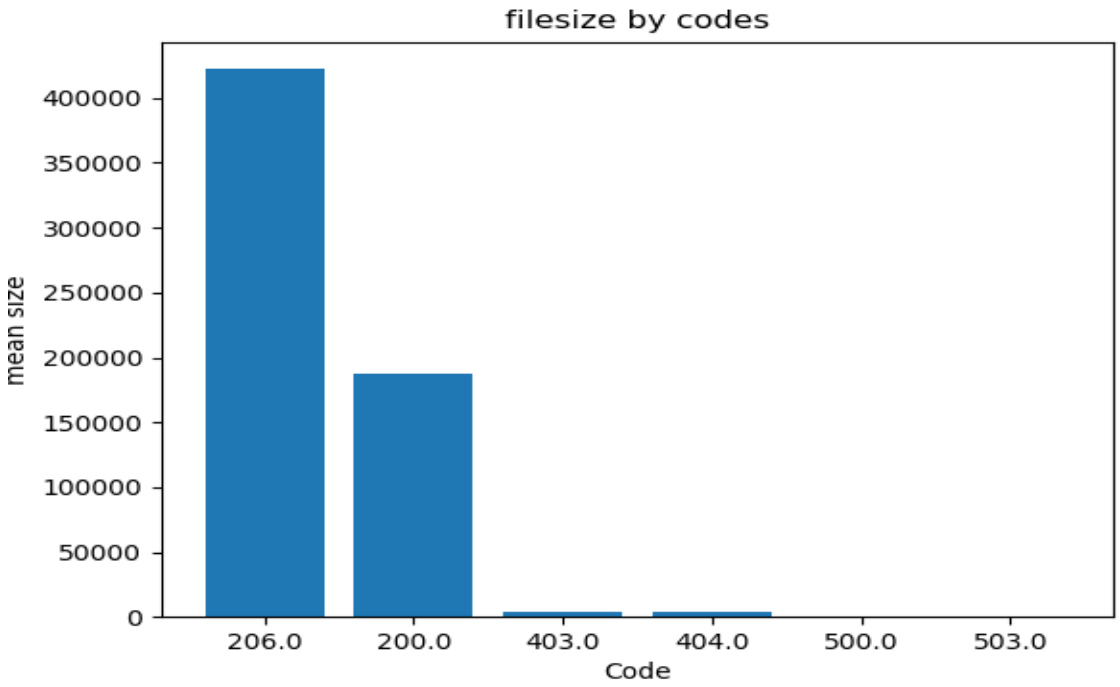




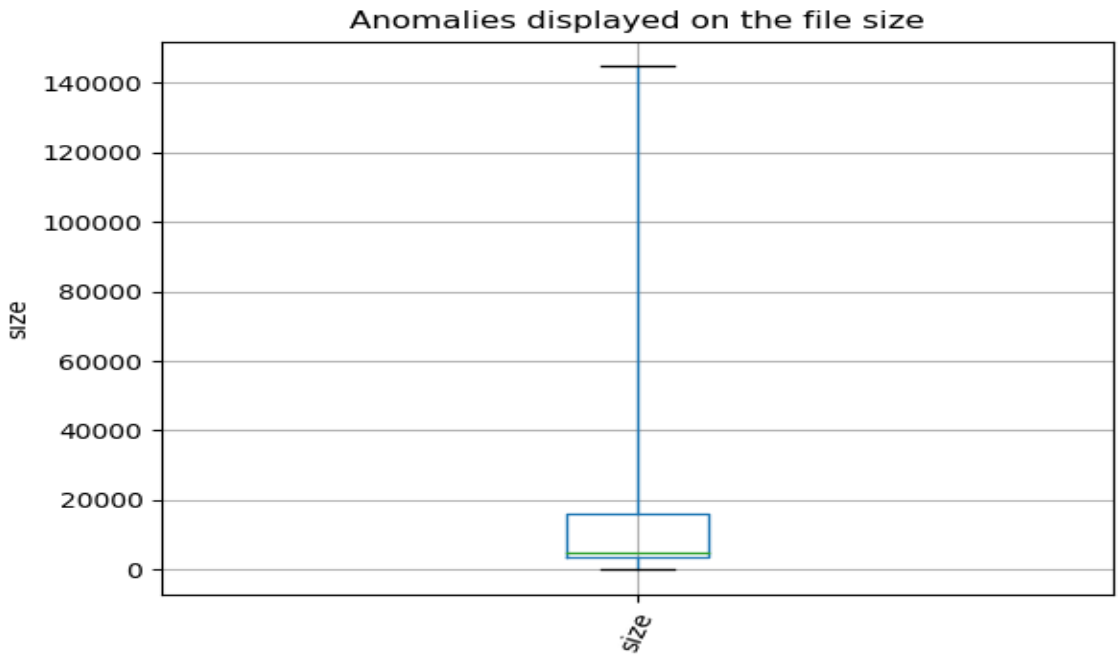
	code	mean
1	206.0	422346.549425
0	200.0	187669.064187
2	403.0	3632.000000
3	404.0	3451.000000
4	500.0	525.000000
5	503.0	323.000000

	extention	mean
35023	gs6904302-fwp.txt	2.294876e+08
35020	gs6569724-fwp.txt	1.445442e+08
2526	a07-21558_25npx.htm	8.108014e+07
50413	v051383_fwp.htm	8.008723e+07
15161	boafuturecomplete.htm	7.264694e+07
20683	d22815.htm	6.887880e+07
20602	d21731.htm	5.974119e+07
5458	a08-7260_2ncsr.htm	3.403204e+07
4354	a08-2249_1posam.htm	3.236706e+07
5417	a08-7101_1nq.htm	3.103957e+07
15331	brd031_0001331971.txt	3.091128e+07
16493	c23411fe497.htm	3.022326e+07
1289	a06-25229_18497.htm	2.947193e+07
44116	p74855nvcsr.htm	2.794462e+07
20684	d22845_wrapper.htm	2.612327e+07
25298	e61297.htm	2.568189e+07
20950	d421743_ex4-1.htm	2.402662e+07
2166	a07-17329_311k.htm	2.348830e+07
27302	exh991to8kabslb_20069.htm	2.348815e+07
45139	psa_exhibits-combined.htm	2.292322e+07


Avg File size by extention
1e8
2.0
1.5
1.0
0.5
0.0
MeanFileSize
Extention


filesize by codes
400000
350000
300000
250000
200000
150000
100000
50000
0
mean size
206.0
200.0
403.0
404.0
500.0
503.0
Code

BOXPLOT ANAMOLY for the SIZE Column of the log set.



RESULTS and PREDICTIONS BASED ON PREVIOUS ANALYSIS

1. **To calculate which class of IP addresses(Class A, Class B , Class C) has done the maximum EDGAR fillings using mode :**
   For the following the maximun IP address are the Class AIps.

2. **Mean and Median sizes for each Browser on the basis of crawler_len :**

browser	size_mean	size_median	crawler_len
fox	1.155820e+05	9346.0	23.0
iem	3.717001e+06	3717001.0	1.0
iph	5.509095e+05	550909.5	2.0
lin	4.606733e+05	15420.5	256.0
mac	3.048609e+05	8109.0	1291.0
mie	3.722820e+05	15433.0	277.0
opr	4.817409e+03	3559.0	22.0
rim	1.322456e+06	312067.5	22.0
win	1.650593e+05	9207.5	159174.0

3. **Compute distinct count of ip per month i.e. per log file**
   Value: 3895

4. **Computing the count of status code on the basis ofip**

```
[2018-02-10 10:23:0
    code   count
0  200.0  160021
1  206.0     956
2  404.0      91
```

5. **Mean of file size on the basis of code status**

```
    code           mean
0  200.0  167240.206804
1  206.0  169654.670502
2  404.0    3451.000000
```

6. **As per the graph (Boxplot) , no significant anomalies are found.**

Docker Steps to build and run the code:

To build and run locally:
Commands for Problem 2 :

1. Create a docker file

```
vim dockerfile
```

2. Build the image locally

```
docker build -f dockerfile -t assignpart2
```

3. Run the image

```
docker run -e year=yr -e accessKey=acckey -e secretKey=seckey -e
location=loc -ti assignpart2
```

4. Tag the image

```
docker tag <imageid>dhanisha/assignpart2:latest
```

5. Push the image to docker hub

```
docker push dhanisha/assignpart2
```

6. To pull the image from hub

```
docker pull dhanisha/assignpart2
```

7. To run the image that is pulled from the hub

```
docker run -e year=yr -e accessKey=acckey -e secretKey=seckey -e
locaiton=loc -tidhanisha/assignpart2
```

Uploading files to Amazon S3 bucket :

```
#-------------------------------------------------------------------------------------------#
#uploading files to amazon s3
'''BucketAlreadyOwnedByYou errors will only be returned outside of the US Standard region.
Inside the US Standard region (i.e. when you don't specify a location constraint), attempting to recreate a b
try:

    buck_name="ads-part2-assignment"

    S3_client = boto3.client('s3', Location, aws_access_key_id= accessKey, aws_secret_access_key=secretKey)

    if Location == 'us-east-1':
            S3_client.create_bucket(
                Bucket=buck_name,
            )
    else:
        S3_client.create_bucket(
                Bucket=buck_name,
                CreateBucketConfiguration={'LocationConstraint': Location},
            )
    logging.info("Connection is successful")
    S3_client.upload_file("Graphical_images", buck_name, "Graphical_images")
    S3_client.upload_file("'Log_file2.log'", buck_name, "'Log_file2.log'")
    logging.info("Files uploaded successfully")
except Exception as e:
    logging.error("Error uploading files to Amazon s3" + str(e))
```

CONCLUSION

We were able to extract the data from the 10-q form for the given cik and accession number and perform data science feature such a data wrangling, data pre-processing , Exploratory Data Analysis , anomaly detection via statistical and graphical methods on log file data for the year provided. We were also able to handle exceptions.

REFERENCES

- https://stackoverflow.com/questions/33388555/unable-to-install-boto3
- https://stackoverflow.com/questions/4770635/s3-error-the-difference-between-the-request-time-and-the-current-time-is-too-la
- https://www.sec.gov/dera/data/edgar-log-file-data-set.html
- https://old.datahub.io/dataset/edgar
- https://docs.python.org/3/library/urllib.parse.html
- https://docs.python.org/3/
- https://boto3.readthedocs.io/en/latest/
- https://docs.docker.com/engine/reference/commandline/build/
- https://docs.docker.com/get-started/
- https://docs.docker.com/engine/reference/commandline/run/
- https://docs.aws.amazon.com/AmazonS3/latest/gsg/CreatingABucket.html
- https://docs.aws.amazon.com/aws-sdk-php/v3/guide/examples/s3-examples.html