Data Handling with Pandas

Contents

Contents

- Importing data
- Accessing
- Sub setting
- Sorting
- Missing values
- Duplicates
- Mergining
- `.apply()`
- Groupby
- Cross tabs and pivots
- Dask

Data import from CSV files

- Need to use the function read.csv
- Need to use “/” or “\\” in the path. The windows style of path “\” doesn’t work

Importing from CSV files

```
import pandas as pd
Sales =
pd.read_csv("https://raw.githubusercontent.com/giridhar276/Data
sets/master/Superstore%20Sales%20Data/Sales_by
_country_v1.csv")
print(Sales)
```

Data import from Excel files

- Need to use pandas again

Data import from Excel files

```
wb_data=
pd.read_excel("https://raw.githubusercontent.com/giridhar276/Datasets/
master/World%20Bank%20Data/GDP.xlsx" , sheet_name=" Sheet1")
print(wb_data)
```

Tip – You can also import zipped files

```
air_bnb_ny=pd.read_csv("https://raw.githubusercontent.com/
giridhar276/Datasets/master/AirBnB_NY/AB_NYC_2019
.zip", compression="zip")


print(air_bnb_ny.shape)
```

Basic Commands on Datasets

- Is the data imported correctly? Are the variables imported in right format? Did we import all the rows?
- Once the dataset is inside Python, we would like to do some basic checks to get an idea on the dataset.
- Just printing the data is not a good option, always.
- Is a good practice to check the number of rows, columns, quick look at the variable structures, a summary and data snapshot

Check list after Import

```
Sales.shape
Sales.info()
Sales.columns
Sales.head()
Sales.tail()
Sales.sample(5)
Sales.describe()
Sales["unitsSold"].describe()
Sales["salesChannel"].value_counts()
```

Lab: Printing the data and meta info

- Import “Credit_Card_Cust_Usage/Card_Usage_v1.csv” data
- How many rows and columns are there in this dataset?
- Print only column names in the dataset
- Print first 10 observations
- Print the last 5 observations
- Take a random sample of 300 records.
- How to reproduce the same random sample again?
- Describe the field Credit_Limit – What are your findings?
- Describe the field Card_Category – What are your findings?
- Create a new dataset by taking first 30 observations from this data
- Print the resultant data
- Remove(delete) the new dataset

Access rows

```
Sales.iloc[0:10]
Sales.iloc[[1,9,10]]
```

Access Columns

```
column_names=['custId', 'custName', 'custCountry']
Sales[column_names]
Sales.iloc[:,0:4]
Sales.iloc[0:5:,0:4]
```

.iloc vs .loc

```
Sales1=Sales.iloc[20:30]
```

What is the difference?

```
Sales1.iloc[0:5]
Sales1.loc[0:5]
```

.iloc vs .loc

```
Sales1=Sales.iloc[20:30]
```

What is the difference?

```
Sales1.iloc[0:5] #index location
Sales1.loc[0:5] #index values or names
```

What is the difference?

```
Sales.loc[0:5, 0:2]
Sales.loc[0:5, column_names]
```

Accessing Specific type of Columns only

```
Sales_numerics = Sales.select_dtypes(include=["int64","float64"])
Sales_numerics.info()

Sales_objects = Sales.select_dtypes(include=["object"])
Sales_objects.info()

Sales_non_objects = Sales.select_dtypes(exclude=["object"])
Sales_non_objects.info()
```

Drop

```
wb_data.drop(range(0,10))
wb_data.drop(["Country_code"], axis=1)
```

Lab: Accessing the data

- Data : "./Bank Tele Marketing/bank_market.csv“. Create separate datasets for each of the below tasks
- Select first 1000 rows only
- Select only four columns "Cust_num" "age” "default" and "balance"
- Select 20,000 to 40,000 observations along with four variables "Cust_num" "job" "marital" and "education"
- Select 5000 to 6000 observations and drop "poutcome“ and "y"
- Access the last column only;(using index)
- Access all the numeric columns only

Subset with variable filter conditions

- How to filter the data based on a variable?
- For example select all the customers with age >40 in bank market data
- Subset all the customers with age>40 and loan=“no”

Subset with variable filter conditions

```
bank_subset=bank_data[bank_data['age']>40]
bank_subset

#And condition & filters
bank_subset1=bank_data[(bank_data['age']>40)  &  (bank_data['loan']=="no")]
bank_subset1

#OR condition & filters
bank_subset2=bank_data[(bank_data['age']>40)  |  (bank_data['loan']=="no")]
bank_subset2
```

Lab: Subset with variable filter conditions

- Data : “./Automobile Data Set/AutoDataset.csv”
- Create a new dataset for exclusively Toyota cars
- Create a new dataset for all cars with city.mpg greater than 30 and engine size is less than 120.
- Create a new dataset by taking only sedan cars. Keep only four variables(Make, body style, fuel type, price) in the final dataset.
- Create a new dataset by taking Audi, BMW or Porsche company makes.

Working with index

```
bank_data.index
bank_subset1.index
```

Reset the index, if required.

```
bank_subset1=bank_subset1.reset_index()
bank_subset1.index
```

The above code creates a new column called index, you can drop it while creating

```
bank_subset1_1=bank_subset1.reset_index(drop=True)
bank_subset1_1.index
```

Calculated Fields

Calculate and Assign it to new variable

```
auto_data['area']=(auto_data[' length'])*(auto_data[' width'])*(auto_data
[' height'])
auto_data['area']
```

Sorting the data

- Its ascending by default

```
Online_Retail_sort=Online_Retail.sort_values('UnitPrice')
Online_Retail_sort.head(20)
```

- Use ascending=False for descending sort

```
Online_Retail_sort=Online_Retail.sort_values('UnitPrice',ascending=False)
Online_Retail_sort.head(20)
```

- Sorting with two cols

```
Online_Retail_sort2=Online_Retail.sort_values(['Country','UnitPrice'], ascen
ding=[True, False])
Online_Retail_sort2.head(5)
```

LAB: Sorting the data

- AutoDataset
- Sort the dataset based on price
- Sort the dataset based on price descending

Identifying & Removing Duplicates

Datasets: Telecom Data Analysis\Bill.csv

```
#Identify duplicates records in the data
dupes=bill_data.duplicated()
dupes
sum(dupes)
```

```
#Removing Duplicates
bill_data.shape
bill_data_uniq=bill_data.drop_duplicates()
bill_data_uniq.shape
```

Identifying & Duplicates based on Key

- What if we are not interested in overall level records
- Sometimes we may name the records as duplicates even if a key variable is repeated.
- Instead of using duplicated function on full data, we use it on one variable

```
dupe_id=bill_data["cust_id"].duplicated()
dupe_id
bill_data.shape
bill_data_cust_uniq=bill_data.drop_duplicates(['cust_id'])
bill_data_cust_uniq.shape
```

LAB: Handling Duplicates

- DataSet: "./Telecom Data Analysis/Complaints.csv"
- Identify overall duplicates in complaints data
- Create a new dataset by removing overall duplicates in Complaints data
- Identify duplicates in complaints data based on cust_id
- Create a new dataset by removing duplicates based on cust_id in Complaints data

Data sets merging and Joining

- Datasets: TV Commercial Slots Analysis/orders.csv & TV Commercial Slots Analysis/slots.csv

```
orders1=orders.drop_duplicates(['Unique_id'])
slots1=slots.drop_duplicates(['Unique_id'])

###Inner Join
inner_data=pd.merge(orders1, slots1, on='Unique_id', how='inner')
inner_data.shape
###Outer Join
outer_data=pd.merge(orders1, slots1, on='Unique_id', how='outer')
outer_data.shape
##Left outer Join
L_outer_data=pd.merge(orders1, slots1, on='Unique_id', how='left')
L_outer_data.shape
###Righ outer Join
R_outer_data=pd.merge(orders1, slots1, on='Unique_id', how='right')
R_outer_data.shape
```

Data sets merging and Joining

- Datasets: TV Commercial Slots Analysis/orders.csv & TV Commercial Slots Analysis/slots.csv

```
orders1=orders.drop_duplicates(['Unique_id'])
slots1=slots.drop_duplicates(['Unique_id'])

###Inner Join
inner_data=pd.merge(orders1, slots1, on='Unique_id', how='inner')
inner_data.shape
###Outer Join
outer_data=pd.merge(orders1, slots1, on='Unique_id', how='outer')
outer_data.shape
##Left outer Join
L_outer_data=pd.merge(orders1, slots1, on='Unique_id', how='left')
L_outer_data.shape
###Righ outer Join
R_outer_data=pd.merge(orders1, slots1, on='Unique_id', how='right')
R_outer_data.shape
```

Data sets merging and Joining

```
####Other options
left_on : a column or a list of columns
right_on : a column or a list of columns
```

```
other_data=pd.merge(orders1, slots1, left_on=['AD_ID','Product ID'],ri
ght_on=['AD_ID','Product ID'], how='outer’)

other_data.shape
```

LAB: Data Joins

- Datasets
  - “./Telecom Data Analysis/Bill.csv”
  - “./Telecom Data Analysis/Complaints.csv”
- Import the data and remove duplicates based on cust_id
- Create a dataset for each of these requirements
  - All the customers who appear either in bill data or complaints data
  - All the customers who appear both in bill data and complaints data
  - All the customers from bill data: Customers who have bill data along with their complaints
  - All the customers from complaints data: Customers who have Complaints data along with their bill info

Functions on rows and columns

- Consider Credit Card usage data.
- Create a new variable credit_limit_segment by using this below condition
  - If Credit_Limit < 3000- Low
  - If Credit_Limit < 6000- medium
  - Else - high

Using a for loop

```
cc_usage["Credit_Limit_Segment"]=0

for i in range(0, len(cc_usage)):
  if cc_usage["Credit_Limit"][i] < 3000:
    cc_usage["Credit_Limit_Segment"][i]  ="Low"
  elif cc_usage["Credit_Limit"][i] < 6000:
    cc_usage["Credit_Limit_Segment"][i]  ="Medium"
  else:
    cc_usage["Credit_Limit_Segment"][i]  ="high"
```

tip- never use a for loop

- For loop code and iteration over each row works, but never use it.
- It is very basic and time consuming
- There are better ways
  - Vectorizing
  - apply function

Vectorization

```
cc_usage["Credit_Limit_Segment1"]="High"
cc_usage["Credit_Limit_Segment1"][cc_usage["Credit_Limit"]< 3000]="Low"
cc_usage["Credit_Limit_Segment1"][(cc_usage["Credit_Limit"]> 3000) & (cc_usa
ge["Credit_Limit"]< 6000)]="Medium"
```

For loop vs Vectorization

- The execution time comparison for the same operation

For loop

```
Time taken 232.56957602500916
```

Vectorization

```
Time taken 0.029158592224121094
```

LAB: For loop vs Vectorization

- Consider Online retail sales data.
- Create a new variable unit_price_segment by using this below condition
  - If unit_price < 2- Low
  - If unit_price < 4- medium
  - Else - high
- Use vectorization- How much time does it take ?
- Use a for loop – How much time does it take?

Apply function

- In the previous examples we worked on two different datasets, and different variables.
- In both the cases, credit limit and UnitPrice, we performed binning with different set of limits.
- Can we write a generalized binning function and then apply it on any column?

Generalized function

```
def binning(x, limit1, limit2):
  result="High"
  if x < limit1:
    result="Low"
  if (x > limit1) & (x < limit2):
    result="Medium"
  return(result)
```

How do we apply the above function on a desired column from a dataset? – Using .apply() function

apply() function

- `cc_usage["Credit_Limit_Segment2"]=cc_usage["Credit_Limit"].apply(lambda x:binning(x, 3000, 6000))`

- Here `lambda` is a temporary anonymous function is made in apply() itself
- `lambda` function takes each row supplies it to the binning function then returns the result of binning function

apply() function

```
cc_usage["Credit_Limit_Segment2"]=cc_usage["Credit_Limit"]
.apply(lambda x:binning(x, 3000, 6000))
```

```
Online_Retail["UnitPrice_Segment2"]=Online_Retail["UnitPri
ce"].apply(lambda x:binning(x, 2, 4))
```

More on .apply() function

- We need not define a function always

```
cc_usage['Customer_Age_new']=cc_usage['Customer_Age'].appl
y(lambda x: "Low" if x<50 else "high"  )
```

.apply() on columns

- How to find the mean of each numeric column in a data frame?
- To work on each column we need to make axis=0. This sounds opposite to drop function
- axis=0 – Apply along the row index (returns for each col)
- axis=1 – Appy along the columns (returns for each row)

```
cc_usage.select_dtypes(exclude="object").apply(lambda y: round(y.mean()),
axis=0)
```

```
cc_usage.select_dtypes(exclude="object").apply(lambda y: [round(y.mean()),
 round(y.median())], axis=0)
```

.apply() on columns

- Applying conditions on multiple columns to make a new column
- Create a new variable based on two conditions

`Total_Relationship_Count > 4 and Credit_Limit < 5000`

```
def cli_flag(x):
  if x[0] > 4 and x[1]<5000:
    return(1)
  else:
    return(0)

cc_usage["Credit_Line_increase_flag"]=cc_usage[["Total_Relations
hip_Count","Credit_Limit"]].apply(lambda x:cli_flag(x), axis=1)
```

LAB: .apply()

- Dataset: Bank Tele marketing data.
- Create a new column “new_bal” by taking the maximum of 0 and balance
- Count the number of missing values in each column by using apply function
- Count the number of missing values in each row by using apply function
- Create a generalized function that can apply on “job” variable
  - If the length of the string is more than 10 then “long”, else “short”.
  - Apply it on one more column

Group-by

- Import House Sales in King County data. https://www.kaggle.com/harlfoxem/housesalesprediction
- group by “condition” of the house and find the below details
  - The number of items in each group
  - The average house price in each group

Group-by

```
kc_house_price.groupby("condition").count()
```

This code gives us the count by each column

```
#Restrict to one column
kc_house_price.groupby("condition")["id"].count()
```

```
round(kc_house_price.groupby("condition")["price"].mean())
```

tip – use agg() function

```
kc_house_price.groupby("condition")["id"].count()
```

The above code works, but its better use the aggregate function. agg() function gives us many options

```
kc_house_price.groupby("condition").agg({'id':['count’]})
```

```
kc_house_price.groupby("condition").agg({'price':['mean']})
```

One group-by but multiple aggregated values

```
kc_house_price_grp_agg=kc_house_price.groupby("condition")
.agg({'id':['count'],'price':['mean','min', 'max'] })
```

Group-by more than one column

```
kc_house_price_grp_agg1=kc_house_price.groupby(["condition","floo
rs"]).agg({'id':['count'],'price':['mean','min', 'max'] })
```

tip -Reset index after group by

- The group-by output gives you multi-indexed columns and rows.
- There is a tendency to use the resultant dataset directly.
- Multi-Index is confusing, better to reset the row index and rename the columns.

```
kc_house_price_grp_agg1.columns
kc_house_price_grp_agg1.index

#Updated index
kc_house_price_grp_agg1.columns=['count','avg_price','min_price',
'max_price']
kc_house_price_grp_agg1=kc_house_price_grp_agg1.reset_index()
```

LAB: Group by and aggregate

- Dataset : Rossmann Store Sales
- https://www.kaggle.com/c/rossmann-store-sales
- Find the average sales on each day of the week.
- Find the median sales and median number of customers across all Assortment types
- For each store type and assortment calculate the
  - Avg , Min and Max sales
  - Avg , Min and Max CompetitionDistance
  - Avg , Min and Max Customers
- Reset the row-index and column names for the above result

Cross-tabs and Pivot tables

- Dataset bank marketing data
- Calculate the response rate in each category of education

```
pd.crosstab(index=bank_data['education'], columns=bank_data['y’])
```

The above code gives the cross tab with counts. For calculation of percentage we need to use apply function on reach row.

```
pd.crosstab(index=bank_data['education'], columns=bank_data['y']).apply(lam
bda x: x*100/x.sum(), axis=1)
```

Cross-tabs and Pivot tables

- Calculate the response rate in each category of job type

```
pd.crosstab(index=bank_data['job'], columns=bank_data['y']
).apply(lambda x: x*100/x.sum(), axis=1)
```

Pivot tables

- Alternative to cross tabs. Many more parameters.
- Calculate the response rate in each category of job type

```
pd.pivot_table(data=bank_data, index='job', columns='y', v
alues='Cust_num', aggfunc='count’)
```

```
pd.pivot_table(data=bank_data, index='job', columns='y', v
alues='Cust_num', aggfunc='count').apply(lambda x: x*100/x
.sum(), axis=1)
```

Pivot tables

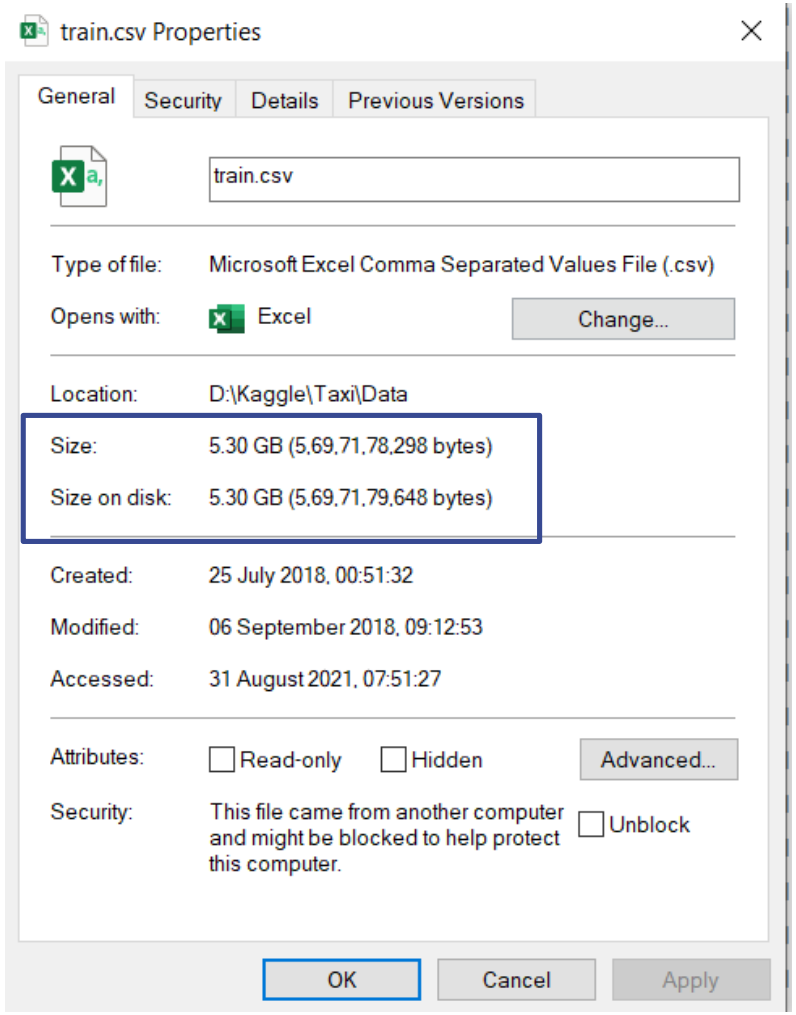
- Create a pivot table by that can calculate the average salary in responder and non-responder category inside each of the job type

```
pd.pivot_table(data=bank_data, index='job', columns='y', v
alues='balance', aggfunc='mean')
```

LAB: Cross tabs and Pivot tables

- Dataset - rossmann_sales
  - Are they any promotions run on weekends?
  - What is the day wise average sales across all the storetypes
- Dataset - cc_usage
  - What is the attrition rate in each card category. Which category has the highest attrition rate ?
  - Compare the average transaction amount of attrition vs existing customers across each card category

Handling Lage Datasets



Handling Lage Dataset

- Is it possible to import this data as a pandas data frame?
- Is it possible to index and store all this data into python?
- Can you import the above dataset.
- Find out number of rows and columns
- Perform basic descriptive statistics.
- Take a random sample of 1 million records from it.

Dask for large datasets

- Pandas data frames are very slow for large datasets(of size in GBs and TBs)
- Dask is a dedicated package to handle large amounts of data
- Dask can store the data on RAM as well as hard-disk
- Dask can store and access the data from multiple machines in a cluster
- Dask will automatically partition and index the data while reading.

Dask for a large dataset

- Dask doesn’t really import the dataset.
- It creates the necessary partitions, pointers and indexes

```
import dask.dataframe as dd
%time taxi_fare_dd = dd.read_csv(r"D:\Kaggle\Taxi\Data\train.csv")
```

```
Wall time: 20.9 ms
```

```
taxi_fare_dd
```

Dask DataFrame Structure:

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
npartitions=90								
	object	float64	object	float64	float64	float64	float64	int64
	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...

Dask Name: read-csv, 90 tasks

Dask for a large dataset

- When we say compute() then the parallel processing will happen to calculate the results.

```
: taxi_fare_dd.shape[0].compute()
```

```
: 55423856
```

Check the background processing status

```
from dask.distributed import Client, progress
client = Client()
client
```

Basic Statistics

```
taxi_fare_dd["fare_amount"].mean().compute()
```

Sampling

```
taxi_fare_dd_sample=taxi_fare_dd.sample(frac=0.02)
```

Converting sample into a dataframe

```
taxi_fare_dd_pd_sample=taxi_fare_dd_sample.compute()
```

```
taxi_fare_dd_pd_sample.shape
```

Conclusion

- In this session we started with Data imploring from various sources
- We also learnt manipulating the datasets and creating new variables
- There are many more topics to discuss in data handling, these topics in the session are essential for any data scientist