Amrita School of Computing

Fifth Semester BTech CSE

19CSE304 Foundations of Data Science

Lab Sheet #1

Exercise 1: PRAACTICE PANDAS LIBRARY ON diabetes DATASET

- **Importing pandas**

To begin working with pandas, import the pandas Python package as shown below. When importing pandas, the most common alias for pandas is pd.

```
import pandas as pd
```

- **Load Dataset**
  - **Mounting Google Drive**

```
from google.colab import drive

drive.mount('/content/drive')
```

- **Importing csv files**

Use read_csv() with the path to the CSV file to read a comma-separated values file.

```
df=pd.read_csv('diabetes.csv')
```

- **Importing Text files**

Reading text files is similar to CSV files. The only nuance is that you need to specify a separator with the sep argument, as shown below.

The separator argument refers to the symbol used to separate rows in a Df.

Comma (sep = ","), whitespace(sep = "\s"), tab (sep = "\t"), and colon(sep = ":") are the commonly used separators. Here \s represents a single white space character.

```
df=pd.read_csv('diabetes.txt',sep="\s")
```

- **Importing Excel files (single sheet)**

Reading excel files (both XLS and XLSX) is as easy as the read_excel() function, using the file path as an input.

df=pd.read_excel('diabetes.xlsx')

- **Importing Excel files (multiple sheets)**

  Reading Excel files with multiple sheets is not that different. You just need to specify one additional argument, sheet_name, where you can either pass a string for the sheet name or an integer for the sheet position (note that Python uses 0-indexing, where the first sheet can be accessed with sheet_name=0)

  df=pd.read_excel('diabetes.xlsx',sheet_name=1)

- **Viewing and understanding Dfs using pandas**

After reading tabular data as a Df, you would need to have a glimpse of the data. You can either view a small sample of the dataset or a summary of the data in the form of summary statistics.

- **How to view data using .head() and .tail()**

You can view the first few or last few rows of a Df using the .head() or .tail() methods, respectively. You can specify the number of rows through the n argument (the default value is 5).

df.head()

df.tail(n=9)

- **Understanding data using .describe()**

The .describe() method prints the summary statistics of all numeric columns, such as count, mean, standard deviation, range, and quartiles of numeric columns.

df.describe()

You can also modify the quartiles using the percentiles argument. Here, for example, we're looking at the 30%, 50%, and 70% percentiles of the numeric columns in Df df.

Df.describe(percentiles=[0,3,0.5,0,7])

You can isolate specific data types in your summary output by using the include argument. Here, for example, we're only summarizing the columns with the integer data type.

df.describe(include=[int])

Try transposing them with the .T attribute.

df.describe().T

- **Understanding data using .info()**

The .info() method is a quick way to look at the data types, missing values, and data size of a DataFrame.. When verbose is set to True, it prints the full summary from .info().

df.info(show_counts=True,memory_usage=True,verbose=True)

- **Understanding your data using .shape**

The number of rows and columns of a DataFrame can be identified using the .shape attribute of the DataFrame. It returns a tuple (row, column) and can be indexe

df.shape # Get the number of rows and columns

df.shape[0] # Get the number of rows only

df.shape[1] # Get the number of columns onlyd to get only rows, and only columns count as output.

- **Get all columns and column names**

Calling the .columns attribute of a DataFrame object returns the column names in the form of an Index object. As a reminder, a pandas index is the address/label of the row or column.

df.columns

- **Checking for missing values in pandas with .isnull()**

The sample DataFrame does not have any missing values. Let's introduce a few to make things interesting. The .copy() method makes a copy of the original DataFrame. This is done to ensure that any changes to the copy don't reflect in the original DataFrame. Using .loc (to be discussed later), you can set rows two to five of the Pregnancies column to NaN values, which denote missing values.

df2 = df.copy()

df2.loc[2:5,'Pregnancies'] = None

df2.head(7)

You can check whether each element in a DataFrame is missing using the .isnull() method.

df2.isnull().head(7)

Given it's often more useful to know how much missing data you have, you can combine .isnull() with .sum() to count the number of nulls in each column.

df2.isnull().sum()

You can also do a double sum to get the total number of nulls in the DataFrame.

df2.isnull().sum().sum()

- **Slicing and Extracting Data in pandas**

The pandas package offers several ways to subset, filter, and isolate data in your DataFrames. Here, we'll see the most common ways.

- **Isolating one column using [ ]**

You can isolate a single column using a square bracket [ ] with a column name in it. The output is a pandas Series object. A pandas Series is a one-dimensional array containing data of any type, including integer, float, string, boolean, python objects, etc. A DataFrame is comprised of many series that act as columns.

```
df['Outcome']
```

- **Isolating two or more columns using [[ ]]**

You can also provide a list of column names inside the square brackets to fetch more than one column. Here, square brackets are used in two different ways. We use the outer square brackets to indicate a subset of a DataFrame, and the inner square brackets to create a list.

```
df[['Pregnancies', 'Outcome']]
```

- **Isolating one row using [ ]**

A single row can be fetched by passing in a boolean series with one True value. In the example below, the second row with index = 1 is returned. Here, .index returns the row labels of the DataFrame, and the comparison turns that into a Boolean one-dimensional array.

```
df[df.index==1]
```

- **Isolating two or more rows using [ ]**

Similarly, two or more rows can be returned using the .isin() method instead of a == operator.

```
df[df.index.isin(range(2,10))]
```

- **Using .loc[] and .iloc[] to fetch rows**

You can fetch specific rows by labels or conditions using .loc[] and .iloc[] ("location" and "integer location"). .loc[] uses a label to point to a row, column or cell, whereas .iloc[] uses the numeric position. To understand the difference between the two, let’s modify the index of df2 created earlier.

```
df2.index = range(1,769)
```

The below example returns a pandas Series instead of a DataFrame. The 1 represents the row index (label), whereas the 1 in .iloc[] is the row position (first row).

```
df2.loc[1]
```

```
df2.iloc[1]
```

You can also fetch multiple rows by providing a range in square brackets.

```
df2.loc[100:110]
```

You can also subset with .loc[] and .iloc[] by using a list instead of a range.

```
df2.loc[[100, 200, 300]]
```

You can also select specific columns along with rows. This is where .iloc[] is different from .loc[] – it requires column location and not column labels.

```
df2.loc[100:110, ['Pregnancies', 'Glucose', 'BloodPressure']]
```

For faster workflows, you can pass in the starting index of a row as a range.

```
df2.loc[760:, ['Pregnancies', 'Glucose', 'BloodPressure']]
```

- **Conditional slicing (that fits certain conditions)**

pandas lets you filter data by conditions over row/column values. For example, the below code selects the row where Blood Pressure is exactly 122. Here, we are isolating rows using the brackets [ ] as seen in previous sections. However, instead of inputting row indices or column names, we are inputting a condition where the column BloodPressure is equal to 122. We denote this condition using df.BloodPressure == 122.

```
df[df.BloodPressure == 122]
```

The below example fetched all rows where Outcome is 1. Here df.Outcome selects that column, df.Outcome == 1 returns a Series of Boolean values determining which Outcomes are equal to 1, then [] takes a subset of df where that Boolean Series is True.

```
df[df.Outcome == 1]
```

You can use a > operator to draw comparisons. The below code fetches Pregnancies, Glucose, and BloodPressure for all records with BloodPressure greater than 100.

```
df.loc[df['BloodPressure'] > 100, ['Pregnancies', 'Glucose', 'BloodPressure’’]]
```

- **Data analysis in pandas**

The main value proposition of pandas lies in its quick data analysis functionality. In this section, we'll focus on a set of analysis techniques you can use in pandas.

- **Summary operators (mean, mode, median)**

As you saw earlier, you can get the mean of each column value using the .mean() method.

df.mean()

- **Create new columns based on existing columns**

pandas provides fast and efficient computation by combining two or more columns like scalar variables. The below code divides each value in the column Glucose with the corresponding value in the Insulin column to compute a new column named Glucose_Insulin_Ratio.

df2['Glucose_Insulin_Ratio'] = df2['Glucose']/df2['Insulin']

df2.head()

- **Counting using .value_counts()**

Often times you'll work with categorical values, and you'll want to count the number of observations each category has in a column. Category values can be counted using the .value_counts() methods. Here, for example, we are counting the number of observations where Outcome is diabetic (1) and the number of observations where the Outcome is non-diabetic (0).

df['Outcome'].value_counts()

- **Aggregating data with .groupby() in pandas**

pandas lets you aggregate values by grouping them by specific column values. You can do that by combining the .groupby() method with a summary method of your choice. The code below displays the mean of each numeric column grouped by Outcome.

df.groupby('Outcome').mean()

.groupby() enables grouping by more than one column by passing a list of column names, as shown below.

df.groupby(['Pregnancies', 'Outcome']).mean()

Any summary method can be used alongside .groupby(), including .min(), .max(), .mean(), .median(), .sum(), .mode(), and more.

Exercise 2: PRACTICE PANDAS EXERCISE FROM

https://www.w3resource.com/python-exercises/pandas/index.php

Exercise 3: Using Wine Quality dataset do the following:

1. Copy the given winequality dataset to your local folder.
2. Load the winequality dataset using pandas
3. Find the size of the dataset.
4. Get the statistical summary of the data
5. Print the first and last five rows
6. Print the first 7 rows of the dataset.
7. Display the first 7 rows of 4th column to 6th column.
8. Select all the rows of the 3rd and 6th column
9. Check the number of null and non-null values in the dataset columns wise.
10. Rename the ph column as potential of hydrogen
11. Create a new column named total_free ratio which is the ratio of the total sulphur dioxide and free sulphurdioxide
12. Aggregate the data based on quality
13. Slice the data frame and create a new one such that the rows containing total_free ratio values less than 2.7 and greater than 3.2
14. Drop second column values from the dataset.
15. Sort the DataFrame first by '' in descending order, then by '' in ascending order
16. Find the sum and the cumulative sum of second attribute
17. Find the minimum and maximum of third attribute
18. Rename columns of a DataFrame
19. Change the order of a DataFrame columns.
20. Write a DataFrame to CSV file using tab separator.
21. Replace all the NaN values with Zero's in a column of a dataframe
22. Divide a data frame in 60:40 ratio.
23. Combine two series into a DataFrame.
24. Find the row for where the value of a first attribute is maximum.
25. Get the datatypes of columns of a DataFrame.