SEABORN

Introduction:

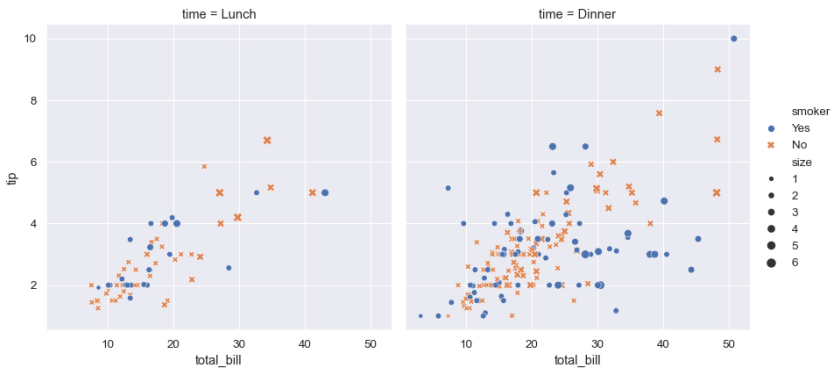
Seaborn is an amazing visualization library for statistical graphics plotting in Python. It provides beautiful default styles and color palettes to make statistical plots more attractive. It is built on top matplotlib library and is also closely integrated with the data structures from pandas.

Seaborn aims to make visualization the central part of exploring and understanding data. It provides dataset-oriented APIs so that we can switch between different visual representations for the same variables for a better understanding of the dataset.

Here's an example of what seaborn can do:

```
# Import seaborn
import seaborn as sns
# Apply the default theme
sns.set_theme()
# Load an example dataset
tips = sns.load_dataset("tips")
# Create a visualization
sns.relplot(
    data=tips,
    x="total_bill", y="tip", col="time",
    hue="smoker", style="smoker", size="size",
)
```

Output:



Different categories of plot in Seaborn:

Plots are basically used for visualizing the relationship between variables. Those variables can be either completely numerical or a category like a group, class, or division.

Seaborn divides the plot into the below categories –

- Relational plots: This plot is used to understand the relation between two variables.
- Categorical plots: This plot deals with categorical variables and how they can be visualized.
- Distribution plots: This plot is used for examining univariate and bivariate distributions
- Regression plots: The regression plots in Seaborn are primarily intended to add a visual guide that helps to emphasize patterns in a dataset during exploratory data analyses.
- Matrix plots: A matrix plot is an array of scatterplots.
- Multi-plot grids: It is a useful approach to draw multiple instances of the same plot on different subsets of the dataset

Loading Dataset:

You must use the popular mtcars dataset for the learning. The data is taken from the 1974 Motor Trend US magazine. It has information about fuel consumption and 10 different aspects of automobile design and performance for 32 cars.

- Let's load this dataset using the Pandas read_csv() function.

```
mtcars = pd.read_csv('C:/Users/avijeet.biswal/Desktop/mtcars.csv')
```

- Below is how the head of the data frame looks like.

```
mtcars.head()
```

	model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

- Now, use the info() function to print the summary of the data frame. It returns information regarding the index dtype, column dtypes, non-null values, and memory usage.

```
mtcars.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32 entries, 0 to 31
Data columns (total 12 columns):
model    32 non-null object
mpg      32 non-null float64
cyl      32 non-null int64
disp     32 non-null float64
hp       32 non-null int64
drat     32 non-null float64
wt       32 non-null float64
qsec     32 non-null float64
vs       32 non-null int64
am       32 non-null int64
gear     32 non-null int64
carb     32 non-null int64
dtypes: float64(5), int64(6), object(1)
memory usage: 3.1+ KB
```

- Now, check the shape of the mtcars dataframe.

```
mtcars.shape
```

```
(32, 12)
```

Seaborn Plotting Functions:

- Styling Plots
- Multiple Plots
- Scatter Plot
- Line Plot
- Bar Plot
- Count Plot
- Box Plot
- Violin Plot
- Strip Plot
- Swarm Plot
- Factor Plot
- Histogram
- Pairplot
- KDE Plot
- Heatmap

Histplot:

Seaborn Histplot is used to visualize the univariate set of distributions(single variable). It plots a histogram, with some other variations like kdeplot and rugplot. The Histplot function takes several arguments but the important ones are

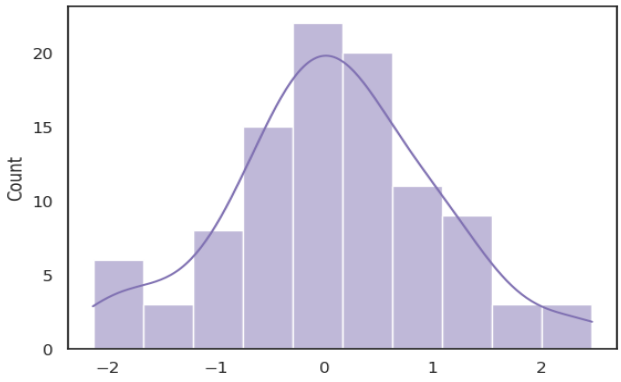
- *data: This is the array, series, or dataframe that you want to visualize. It is a required parameter.*
- *x: This specifies the column in the data to use for the histogram. If your data is a dataframe, you can specify the column by name.*
- *y: This specifies the column in the data to use for the histogram when you want to create a bivariate histogram. By default, it is set to None, meaning that a univariate histogram will be plotted.*

- *bins: This specifies the number of bins to use when dividing the data into intervals for plotting. By default, it is set to "auto", which uses an algorithm to determine the optimal number of bins.*
- *kde: This parameter controls whether to display a kernel density estimate (KDE) of the data in addition to the histogram. By default, it is set to False, meaning that a KDE will not be plotted.*

Example:

```
import numpy as np
import seaborn as sns
sns.set(style="white")
# Generate a random univariate dataset
rs = np.random.RandomState(10)
d = rs.normal(size=100)
# Plot a simple histogram and kde
ns.histplot(d, kde=True, color="m")
```

Output:



Distplot:

Seaborn distplot is used to visualize the univariate set of distributions(Single features) and plot the histogram with some other variations like kdeplot and rugplot.

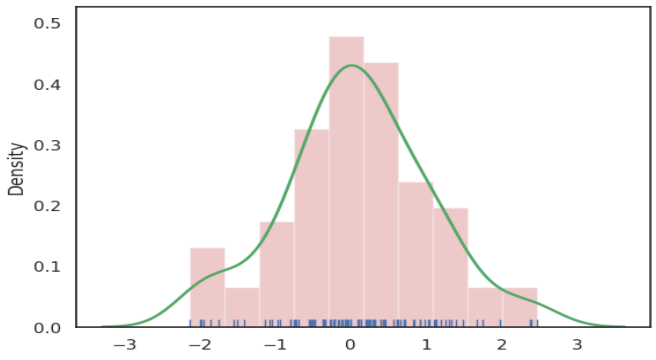
The function takes several parameters, but the most important ones are:

- **a:** *This is the array, series, or list of data that you want to visualize. It is a required parameter.*
- **bins:** *This specifies the number of bins to use when dividing the data into intervals for plotting. By default, it is set to "auto", which uses an algorithm to determine the optimal number of bins.*
- **kde:** *This parameter controls whether to display a kernel density estimate (KDE) of the data in addition to the histogram. By default, it is set to True, meaning that a KDE will be plotted.*
- **hist***: This parameter controls whether to display the histogram of the data. By default, it is set to True, meaning that a histogram will be plotted.*

Example:

```
import numpy as np
import seaborn as sns
sns.set(style="white")
# Generate a random univariate dataset
rs = np.random.RandomState(10)
d = rs.normal(size=100)
# Define the colors to use
colors = ["r", "g", "b"]
# Plot a histogram with multiple colors
sns.distplot(d, kde=True, hist=True, bins=10,
        rug=True,hist_kws={"alpha": 0.3,
                "color": colors[0]},
        kde_kws={"color": colors[1], "lw": 2},
        rug_kws={"color": colors[2]})
```

Output:



Lineplot:

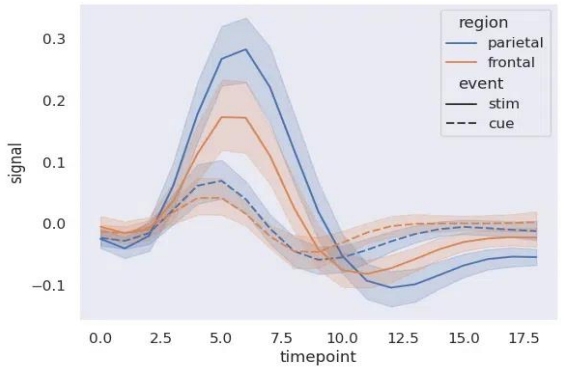
The line plot is one of the most basic plots in the seaborn library. This plot is mainly used to visualize the data in the form of some time series, i.e. in a continuous manner.

Example:

```
import seaborn as sns
sns.set(style="dark")
fmri = sns.load_dataset("fmri")
```

```
# Plot the responses for different
# events and regions
sns.lineplot(x="timepoint",
             y="signal",
             hue="region",
             style="event",
             data=fmri)
```

Output:



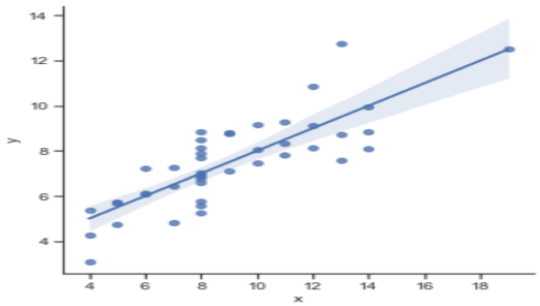
Lmplot:

The lmplot is another most basic plot. It shows a line representing a linear regression model along with data points on the 2D space and x and y can be set as the horizontal and vertical labels respectively.

Example:

```
import seaborn as sns
sns.set(style="ticks")
# Loading the dataset
df = sns.load_dataset("anscombe")
# Show the results of a linear regression
sns.lmplot(x="x", y="y", data=df)
```

output:



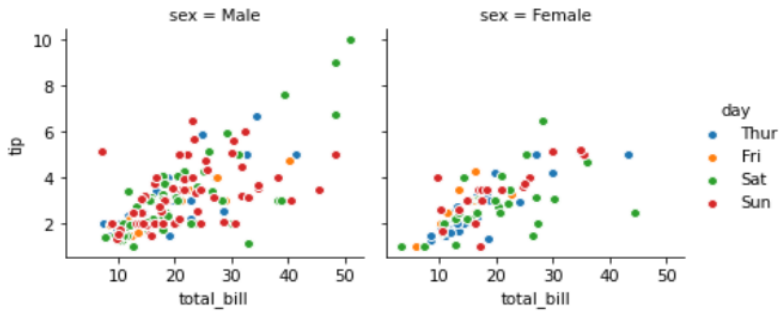
Seaborn.FacetGrid() :

- FacetGrid class helps in visualizing distribution of one variable as well as the relationship between multiple variables separately within subsets of your dataset using multiple panels.
- A FacetGrid can be drawn with up to three dimensions ? row, col, and hue. The first two have obvious correspondence with the resulting array of axes; think of the hue variable as a third dimension along a depth axis, where different levels are plotted with different colors.
- FacetGrid object takes a dataframe as input and the names of the variables that will form the row, column, or hue dimensions of the grid. The variables should be categorical and the data at each level of the variable will be used for a facet along that axis.

Example:

```
 # importing packages

import seaborn

import matplotlib.pyplot as plt

# loading of a dataframe from seaborn

df = seaborn.load_dataset('tips')

# Form a facetgrid using columns with a hue

graph = seaborn.FacetGrid(df, col ="sex", hue ="day")

# map the above form facetgrid with some attributes

graph.map(plt.scatter, "total_bill", "tip", edgecolor ="w").add_legend()

# show the object

plt.show()
```

Output:



Seaborn.PairGrid() :

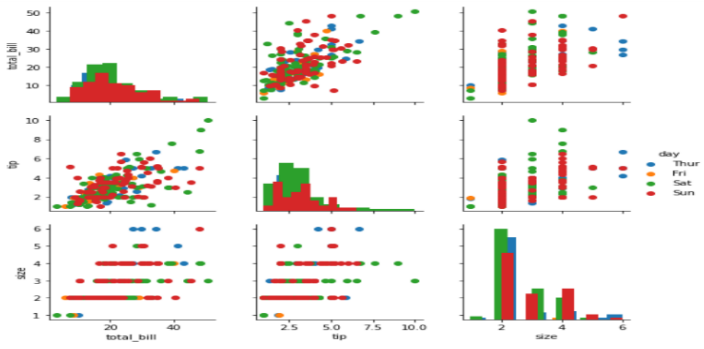
- Subplot grid for plotting pairwise relationships in a dataset.
- This class maps each variable in a dataset onto a column and row in a grid of multiple axes. Different axes-level plotting functions can be used to draw bivariate plots in the upper and lower triangles, and the marginal distribution of each variable can be shown on the diagonal.

- It can also represent an additional level of conditionalization with the hue parameter, which plots different subsets of data in different colors. This uses color to resolve elements on a third dimension, but only draws subsets on top of each other and will not tailor the hue parameter for the specific visualization the way that axes-level functions that accept hue will.

Example :

```
 # importing packages

import seaborn

import matplotlib.pyplot as plt

# loading dataset

df = seaborn.load_dataset('tips')

# PairGrid object with hue

graph = seaborn.PairGrid(df, hue ='day')

# type of graph for diagonal

graph = graph.map_diag(plt.hist)

# type of graph for non-diagonal

graph = graph.map_offdiag(plt.scatter)

# to add legends

graph = graph.add_legend()

# to show

plt.show()
```

Output:



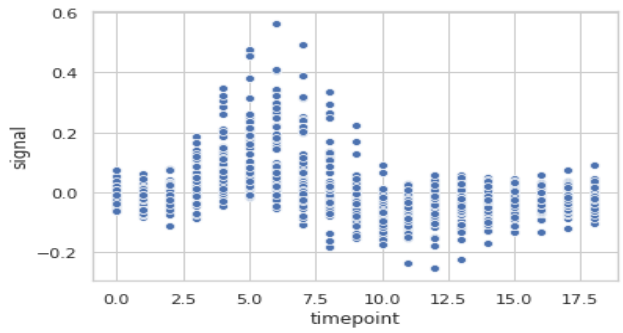
Scatterplot:

Scatterplot can be used with several semantic groupings which can help to understand well in a graph. They can plot two-dimensional graphics that can be enhanced by mapping up to three additional variables while using the semantics of hue, size, and style parameters. All the parameter control visual semantic which are used to identify the different subsets. Using redundant semantics can be helpful for making graphics more accessible.

Example:

```
import seaborn

seaborn.set(style='whitegrid')

fmri = seaborn.load_dataset("fmri")

seaborn.scatterplot(x="timepoint",
                    y="signal",
                    data=fmri)
```

Output:



Seaborn Line Plot:

Single Line Plot

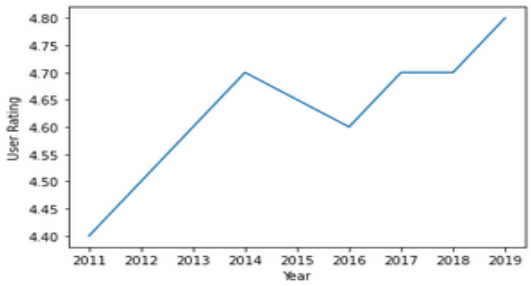
A single line plot presents data on x-y axis using a line joining datapoints. To obtain a graph Seaborn comes with an inbuilt function to draw a line plot called lineplot().

Example:

```
 # import modules

import seaborn as sn

import matplotlib.pyplot as plt

import pandas as pd

# import data

data = pd.read_csv("C:\\Users\\Vanshi\\Desktop\\gfg\\bestsellers.csv")
```

```
# selecting required rows and columns
data = data.iloc[2:10, :]
# plotting a single line graph
sn.lineplot(x="Year", y="User Rating", data=data)
# displaying the plot
plt.show()
```

Output:



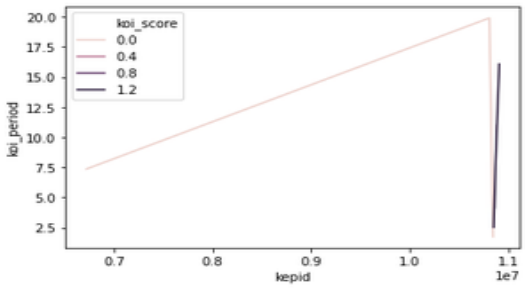
Multiple Line Plot

Functionalities at times dictate data to be compared against one another and for such cases a multiple plot can be drawn. A multiple line plot helps differentiate between data so that it can be studied and understood with respect to some other data. Each lineplot basically follows the concept of a single line plot but differs on the way it is presented on the screen. Lineplot of each data can be made different by changing its color, line style, size or all listed, and a scale can be used to read it.

Example:

```
# import modules
import seaborn as sn
import matplotlib.pyplot as plt
import pandas as pd
# import data
data = pd.read_csv("C:\\Users\\Vanshi\\Desktop\\gfg\\cumulative.csv")
# select required data
data = data.iloc[2:10, :]
# plot data with different color scheme
sn.lineplot(x="kepid", y="koi_period", data=data, hue="koi_score")
# display plot
plt.show()
```

Output:



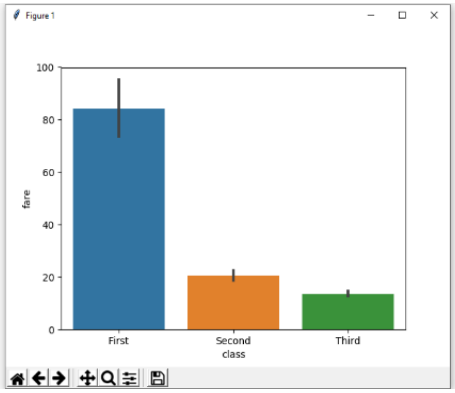
Seaborn.barplot()

Seaborn.barplot() method is used to draw a barplot. A bar plot represents an estimate of central tendency for a numeric variable with the height of each rectangle and provides some indication of the uncertainty around that estimate using error bars.

Example:

```
# importing the required library

import seaborn as sns

import matplotlib.pyplot as plt

# read a titanic.csv file

# from seaborn library

df = sns.load_dataset('titanic')

# class v / s fare barplot

sns.barplot(x = 'class', y = 'fare', data = df)

# Show the plot

plt.show()
```

Output:



Countplot

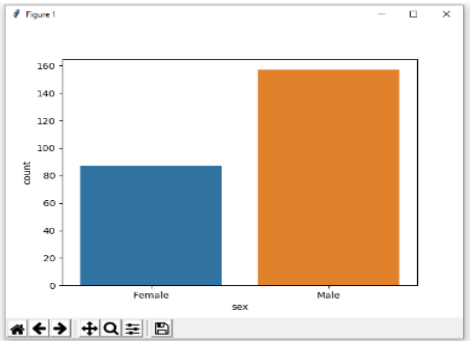
Seaborn.countplot()

Seaborn.countplot() method is used to Show the counts of observations in each categorical bin using bars.

Example:

```
# importing the required library
import seaborn as sns
import matplotlib.pyplot as plt
# read a tips.csv file from seaborn library
df = sns.load_dataset('tips')
# count plot on single categorical variable
sns.countplot(x ='sex', data = df)
# Show the plot
plt.show()
```

Output:



Box Plot

A box plot helps to maintain the distribution of quantitative data in such a way that it facilitates the comparisons between variables or across levels of a categorical variable. The main body of the box plot showing the quartiles and the median's confidence intervals if enabled. The medians have horizontal lines at the median of each box and while whiskers have the vertical lines extending to the most extreme, non-outlier data points and caps are the horizontal lines at the ends of the whiskers.

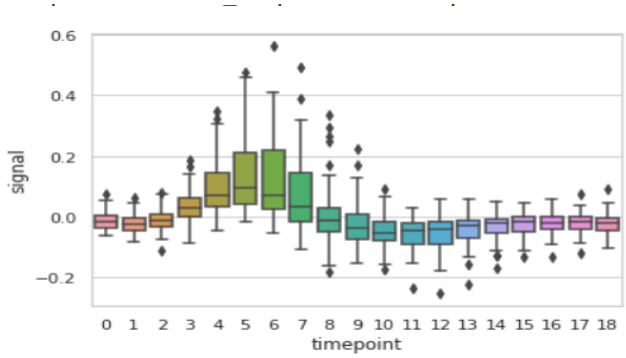
Example:

```
import seaborn
seaborn.set(style='whitegrid')
fmri = seaborn.load_dataset("fmri")
seaborn.boxplot(x="timepoint",
```

y="signal",

data=fmri)

Output:



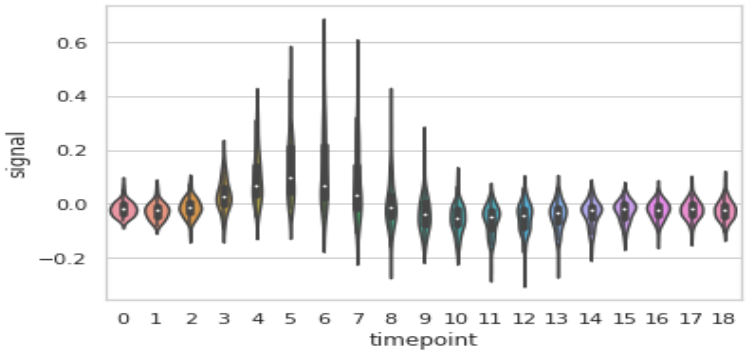
Violin Plot

A violin plot plays a similar activity that is pursued through whisker or box plot do. As it shows several quantitative data across one or more categorical variables. It can be an effective and attractive way to show multiple data at several units. A “wide-form” Data Frame helps to maintain each numeric column which can be plotted on the graph. It is possible to use NumPy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes.

Example:

```
import seaborn
seaborn.set(style = 'whitegrid')
fmri = seaborn.load_dataset("fmri")
seaborn.violinplot(x ="timepoint",
                   y ="signal",
                   data = fmri)
```

Output:



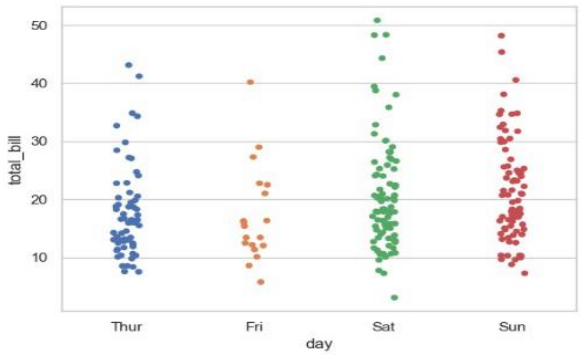
Strip plot

A strip plot is drawn on its own. It is a good complement to a boxplot or violinplot in cases where all observations are shown along with some representation of the underlying distribution. It is used to draw a scatter plot based on the category.

Example:

```
import seaborn

import matplotlib.pyplot as plt

seaborn.set(style = 'whitegrid')

tip = seaborn.load_dataset("tips")

seaborn.stripplot(x="day", y="total_bill", data=tip)

plt.show()
```

Output:



Swarmplot

Seaborn.swarmplot()

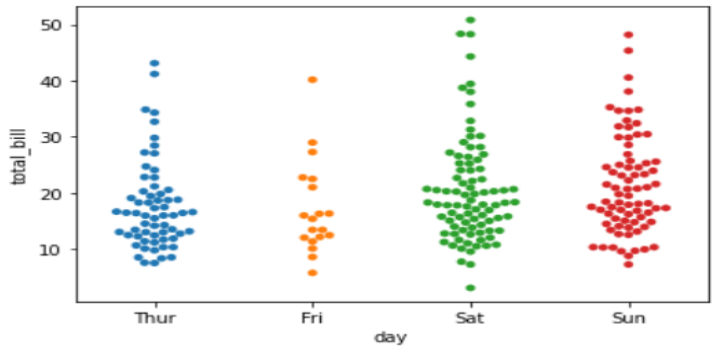
Draw a categorical scatterplot with non-overlapping points. A swarm plot can be drawn on its own, but it is also a good complement to a box or violin plot in cases where you want to show all observations along with some representation of the underlying distribution. Arranging the points properly requires an accurate transformation between data and point coordinates. This means that non-default axis limits must be set *before* drawing the plot.

Example:

```
# importing packages

import seaborn as sns

import matplotlib.pyplot as plt

# loading dataset

data = sns.load_dataset("tips")
```

```
# plot the swarmplot
# size set to 5
sns.swarmplot(x ="day", y = "total_bill",
                                data = data, size = 5)
plt.show()
```

Output:



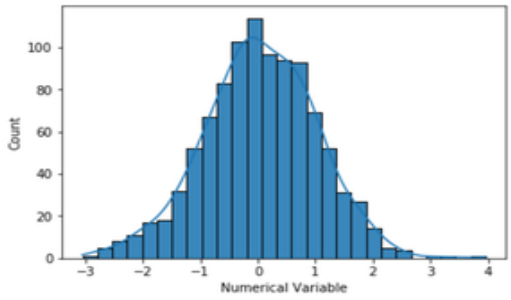
Histogram

Histograms are visualization tools that represent the distribution of a set of continuous data. In a histogram, the data is divided into a set of intervals or bins (usually on the x-axis) and the count of data points that fall into each bin corresponding to the height of the bar above that bin. These bins may or may not be equal in width but are adjacent (with no gaps).

Example:

```
# Import necessary libraries
import seaborn as sns
import numpy as np
import pandas as pd
# Generating dataset of random numbers
np.random.seed(1)
num_var = np.random.randn(1000)
num_var = pd.Series(num_var, name = "Numerical Variable")
# Plot histogram
sns.histplot(data = num_var, kde = True)
```

Output:



Pairplot

Seaborn.pairplot() :

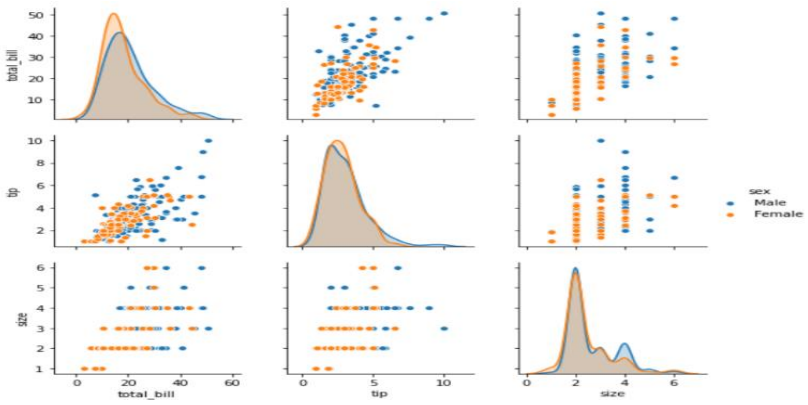
To plot multiple pairwise bivariate distributions in a dataset, you can use the *.pairplot()* function.

The diagonal plots are the univariate plots, and this displays the relationship for the (n, 2) combination of variables in a DataFrame as a matrix of plots.

Example:

```
# importing packages
import seaborn
import matplotlib.pyplot as plt
# loading dataset using seaborn
df = seaborn.load_dataset('tips')
# pairplot with hue sex
seaborn.pairplot(df, hue ='sex')
# to show
plt.show()
```

Output:



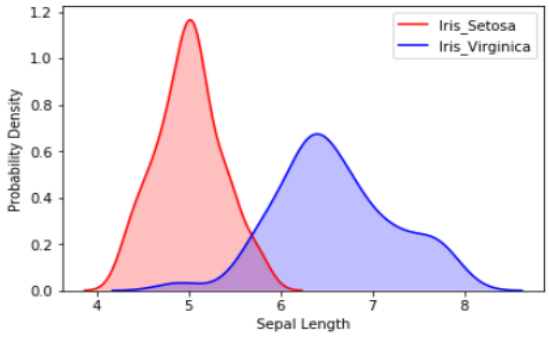
KDE Plot

KDE Plot described as Kernel Density Estimate is used for visualizing the Probability Density of a continuous variable. It depicts the probability density at different values in a continuous variable. We can also plot a single graph for multiple samples which helps in more efficient data visualization. It provides a smoothed representation of the underlying distribution of a dataset.

Example:

```
 # Plotting the KDE Plot
sns.kdeplot(iris_df.loc[(iris_df['Target']=='Iris_Setosa'),
        'Sepal_Length'], color='r', shade=True, label='Iris_Setosa')
sns.kdeplot(iris_df.loc[(iris_df['Target']=='Iris_Virginica'),
        'Sepal_Length'], color='b', shade=True, label='Iris_Virginica')
plt.xlabel('Sepal Length')
plt.ylabel('Probability Density')
```

Output:



Heatmap:

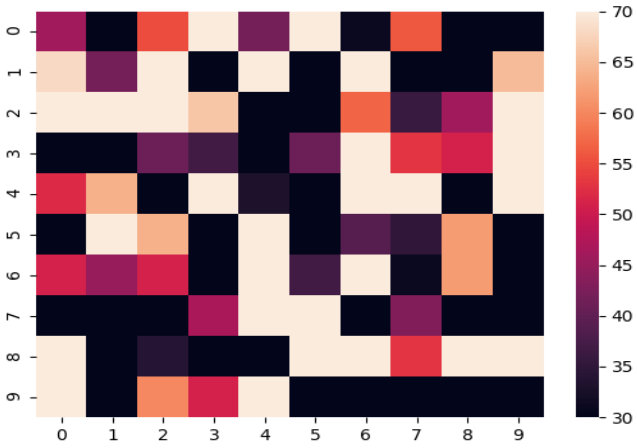
Heatmap is defined as a graphical representation of data using colors to visualize the value of the matrix. In this, to represent more common values or higher activities brighter colors basically reddish colors are used and to represent less common or activity values, darker colors are preferred. Heatmap is also defined by the name of the shading matrix. Heatmaps in Seaborn can be plotted by using the seaborn.heatmap() function.

Example:

```
# importing the modules
import numpy as np
import seaborn as sn
```

```
import matplotlib.pyplot as plt
# generating 2-D 10x10 matrix of random numbers
# from 1 to 100
data = np.random.randint(low=1,
                         high=100,
                         size=(10, 10))
# setting the parameter values
vmin = 30
vmax = 70
# plotting the heatmap
hm = sn.heatmap(data=data,
                vmin=vmin,
                vmax=vmax)
# displaying the plotted heatmap
plt.show()
```

Output:



Conclusion

Data visualization plays an important role in exploratory data analysis and the Seaborn library makes that task really easy and interesting by providing in-built plotting functions, especially for those working with statistical data. Its intuitive interface, built-in themes, and ability to seamlessly integrate with Pandas make it a favorite among data scientists and analysts. With a variety of plotting functions and advanced features like faceting and automatic statistical estimation, Seaborn simplifies the process of creating informative and aesthetically pleasing visualizations. Whether you are exploring data trends or presenting findings, Seaborn enhances your ability to communicate insights effectively.

Pandas

Introduction:

Pandas is an open-source data analysis and manipulation library for Python, designed to provide flexible and efficient data structures for handling structured data. Its primary components are:

- Series: A one-dimensional labeled array capable of holding various data types, similar to a single column in a table.
- DataFrame: A two-dimensional labeled data structure with columns that can hold different types of data, analogous to a spreadsheet or SQL table.

Pandas offers a wide range of functions for data manipulation, including filtering, grouping, merging, and handling missing values. It supports various data formats for input and output, making it a fundamental tool for data analysis, data cleaning, and preparation in Python.

What is pandas?

Pandas is a data manipulation package in Python for tabular data. That is, data in the form of rows and columns, also known as DataFrames. Intuitively, you can think of a DataFrame as an Excel sheet.

pandas' functionality includes data transformations, like sorting rows and taking subsets, to calculating summary statistics such as the mean, reshaping DataFrames, and joining DataFrames together. pandas works well with other popular Python data science packages, often called the PyData ecosystem, including

- NumPy for numerical computing
- Matplotlib, Seaborn, Plotly, and other data visualization packages
- scikit-learn for machine learning

What is pandas used for?

pandas is used throughout the data analysis workflow. With pandas, you can:

- Import datasets from databases, spreadsheets, comma-separated values (CSV) files, and more.
- Clean datasets, for example, by dealing with missing values.
- Tidy datasets by reshaping their structure into a suitable format for analysis.
- Aggregate data by calculating summary statistics such as the mean of columns, correlation between them, and more.
- Visualize datasets and uncover insights.

pandas also contains functionality for time series analysis and analyzing text data.

Install pandas

Installing pandas is straightforward; just use the pip install command in your terminal.

```
pip install pandas
```

Importing CSV files

Use read_csv() with the path to the CSV file to read a comma-separated values file (see our tutorial on importing data with read_csv() for more detail).

```
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 DataFrame. 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)

```
# Extracting the second sheet since Python uses 0-indexing
df = pd.read_excel('diabetes_multi.xlsx', sheet_name=1)
```

Importing JSON file

Similar to the read_csv() function, you can use read_json() for JSON file types with the JSON file name as the argument (for more detail read this tutorial on importing JSON and HTML data into pandas). The below code reads a JSON file from disk and creates a DataFrame object df.

```
df = pd.read_json("diabetes.json")
```

Outputting a DataFrame into a CSV file

A pandas DataFrame (here we are using df) is saved as a CSV file using the .to_csv() method. The arguments include the filename with path and index – where index = True implies writing the DataFrame's index.

```
df.to_csv("diabetes_out.csv", index=False)
```

Outputting a DataFrame into a JSON file

Export DataFrame object into a JSON file by calling the .to_json() method.

```
df.to_json("diabetes_out.json")
```

Outputting a DataFrame into a text file

As with writing DataFrames to CSV files, you can call .to_csv(). The only differences are that the output file format is in .txt, and you need to specify a separator using the sep argument.

```
df.to_csv('diabetes_out.txt', header=df.columns, index=None, sep=' ')
```

Outputting a DataFrame into an Excel file

Call .to_excel() from the DataFrame object to save it as a ".xls" or ".xlsx" file.

```
df.to_excel("diabetes_out.xlsx", index=False)
```

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

You can view the first few or last few rows of a DataFrame 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()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

First five rows of the DataFrame

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()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Get summary statistics with .describe()

data types using exclude argument.

df.describe(exclude=[int])

	BMI	DiabetesPedigreeFunction
count	768.000000	768.000000
mean	31.992578	0.471876
std	7.884160	0.331329
min	0.000000	0.078000
25%	27.300000	0.243750
50%	32.000000	0.372500
75%	36.600000	0.626250
max	67.100000	2.420000

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. Here, we're setting the show_counts argument to True, which gives a few over the total non-missing values in each column. We're also setting memory_usage to True, which shows the total memory usage of the DataFrame elements. When verbose is set to True, it prints the full summary from .info().

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

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
```

```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

It can be converted to a list using a list() function.

```
list(df.columns)
```

```
['Pregnancies',
 'Glucose',
 'BloodPressure',
 'SkinThickness',
 'Insulin',
 'BMI',
 'DiabetesPedigreeFunction',
 'Age',
 'Outcome']
```

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)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6.0	148	72	35	0	33.6	0.627	50	1
1	1.0	85	66	29	0	26.6	0.351	31	0
2	NaN	183	64	0	0	23.3	0.672	32	1
3	NaN	89	66	23	94	28.1	0.167	21	0
4	NaN	137	40	35	168	43.1	2.288	33	1
5	NaN	116	74	0	0	25.6	0.201	30	0
6	3.0	78	50	32	88	31.0	0.248	26	1

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']
```

```
0       1
1       0
2       1
3       0
4       1
      ..
763     0
764     0
765     0
766     1
767     0
Name: Outcome, Length: 768, dtype: int64
```

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']]
```

	Pregnancies	Outcome
0	6	1
1	1	0
2	8	1
3	1	0
4	0	1
...	...	...
763	10	0
764	2	0
765	5	0
766	1	1
767	1	0

768 rows × 2 columns

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]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
1	1	85	66	29	0	26.6	0.351	31	0

Cleaning data using pandas

Data cleaning is one of the most common tasks in data science. pandas lets you preprocess data for any use, including but not limited to training machine learning and deep learning models. Let's use the DataFrame df2 from earlier, having four missing values, to illustrate a few data cleaning use cases. As a reminder, here's how you can see how many missing values are in a DataFrame.

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

Dealing with missing data technique

Dropping missing values

The axis argument lets you specify whether you are dropping rows, or columns, with missing values. The default axis removes the rows containing NaNs. Use axis = 1 to remove the columns with one or more NaN values. Also, notice how we are using the argument inplace=True which lets you skip saving the output of .dropna() into a new DataFrame.

```
df3 = df2.copy()

df3.dropna(inplace=True, axis=1)

df3.head()
```

	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DPF	Age	Outcome	STF
0	148	72	35	0	33.6	0.627	50	1	0.700000
1	85	66	29	0	26.6	0.351	31	0	0.935484
2	183	64	0	0	23.3	0.672	32	1	0.000000
3	89	66	23	94	28.1	0.167	21	0	1.095238
4	137	40	35	168	43.1	2.288	33	1	1.060606

Dropping missing data in pandas

Dealing with Duplicate Data

Let's add some duplicates to the original data to learn how to eliminate duplicates in a DataFrame. Here, we are using the .concat() method to concatenate the rows of the df2 DataFrame to the df2 DataFrame, adding perfect duplicates of every row in df2.

```
df3 = pd.concat([df2, df2])

df3.shape
```

Renaming columns

A common data cleaning task is renaming columns. With the .rename() method, you can use columns as an argument to rename specific columns. The below code shows the dictionary for mapping old and new column names.

```
df3.rename(columns = {'DiabetesPedigreeFunction':'DPF'}, inplace = True)

df3.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DPF	Age	Outcome
0	6.0	148	72	35	0	33.6	0.627	50	1
1	1.0	85	66	29	0	26.6	0.351	31	0
2	8.0	183	64	0	0	23.3	0.672	32	1
3	1.0	89	66	23	94	28.1	0.167	21	0
4	0.0	137	40	35	168	43.1	2.288	33	1

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()
```

```
Pregnancies                 3.845052
Glucose                   120.894531
BloodPressure              69.105469
SkinThickness              20.536458
Insulin                    79.799479
BMI                        31.992578
DiabetesPedigreeFunction    0.471876
Age                        33.240885
Outcome                     0.348958
dtype: float64
```

Printing the mean of columns in pandas

A mode can be computed similarly using the .mode() method.

```
df.mode()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	1.0	99	70.0	0.0	0.0	32.0	0.254	22.0	0.0
1	NaN	100	NaN	NaN	NaN	NaN	0.258	NaN	NaN

Printing the mode of columns in pandas

Similarly, the median of each column is computed with the .median() method

```
df.median()
```

```
Pregnancies                 3.0000
Glucose                   117.0000
BloodPressure              72.0000
SkinThickness              23.0000
Insulin                    30.5000
BMI                        32.0000
DiabetesPedigreeFunction    0.3725
Age                        29.0000
Outcome                     0.0000
dtype: float64
```

printing the median of columns in pandas

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()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome	Glucose_Insulin_Ratio
1	6.0	148	72	35	0	33.6	0.627	50	1	inf
2	1.0	85	66	29	0	26.6	0.351	31	0	inf
3	NaN	183	64	0	0	23.3	0.672	32	1	inf
4	NaN	89	66	23	94	28.1	0.167	21	0	0.946809
5	NaN	137	40	35	168	43.1	2.288	33	1	0.815476

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()
```

```
0    500
1    268
Name: Outcome, dtype: int64
```

Using .value_counts() *in pandas*

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 below code displays the mean of each of the numeric columns grouped by Outcome.

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

Outcome	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	3.298000	109.980000	68.184000	19.664000	68.792000	30.304200	0.429734	31.190000
1	4.865672	141.257463	70.824627	22.164179	100.335821	35.142537	0.550500	37.067164

Aggregating data by one column in pandas

Pivot tables

Pandas also enables you to calculate summary statistics as pivot tables. This makes it easy to draw conclusions based on a combination of variables. The below code picks the rows as unique values of Pregnancies, the column values are the unique values of Outcome, and the cells contain the average value of BMI in the corresponding group.

For example, for Pregnancies = 5 and Outcome = 0, the average BMI turns out to be 31.1.

```
pd.pivot_table(df, values="BMI", index='Pregnancies',
               columns=['Outcome'], aggfunc=np.mean)
```

Outcome	0	1
Pregnancies		
0	31.727397	39.213158
1	29.616038	37.793103
2	29.679762	34.578947
3	29.231250	32.548148
4	31.255556	33.873913
5	31.100000	36.780952
6	29.591176	31.775000
7	29.975000	34.756000
8	30.693750	32.204545
9	28.840000	33.300000
10	30.114286	31.380000
11	37.125000	39.385714
12	30.560000	34.575000
13	33.280000	36.720000
14	NaN	35.100000
15	NaN	37.100000
17	NaN	40.900000

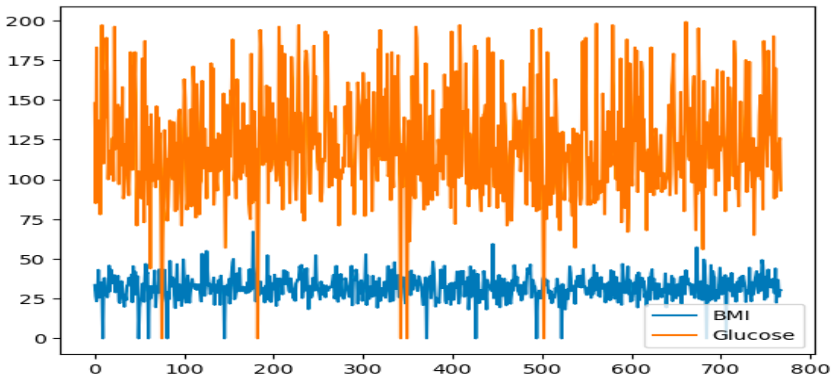
Data visualization in pandas

Pandas provides convenience wrappers to Matplotlib plotting functions to make it easy to visualize your DataFrames. Below, you'll see how to do common data visualizations using pandas.

Line plots in pandas

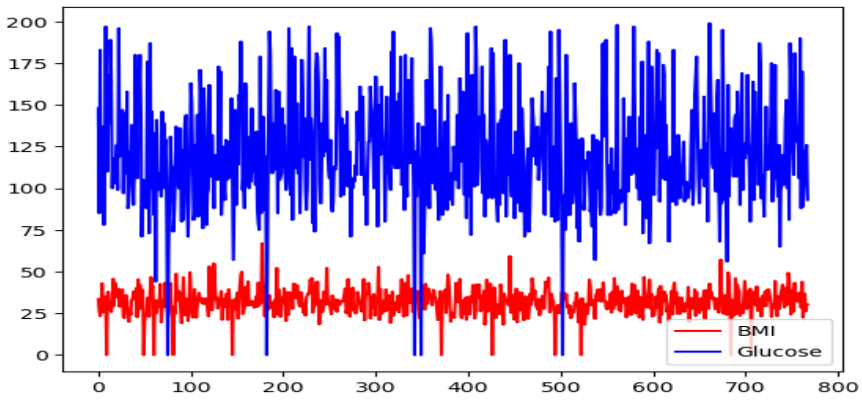
Pandas enables you to chart out the relationships among variables using line plots. Below is a line plot of BMI and Glucose versus the row index.

```
df[['BMI', 'Glucose']].plot.line()
```



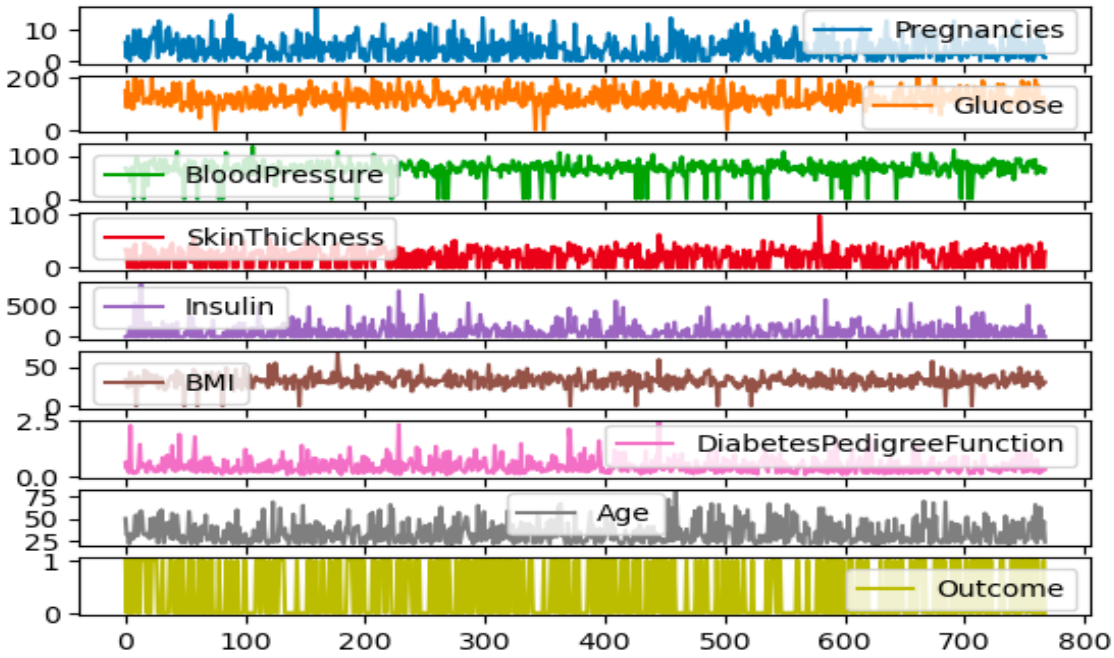
the choice of colors by using the color argument.

```
df[['BMI', 'Glucose']].plot.line(figsize=(20, 10),
                                 color={"BMI": "red", "Glucose": "blue"})
```



All the columns of df can also be plotted on different scales and axes by using the subplots argument.

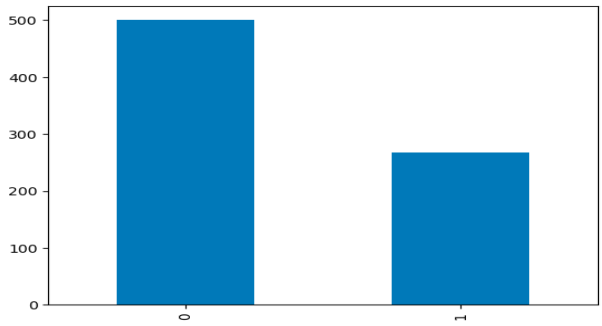
```
df.plot.line(subplots=True)
```



Bar plots in pandas

For discrete columns, you can use a bar plot over the category counts to visualize their distribution. The variable Outcome with binary values is visualized below.

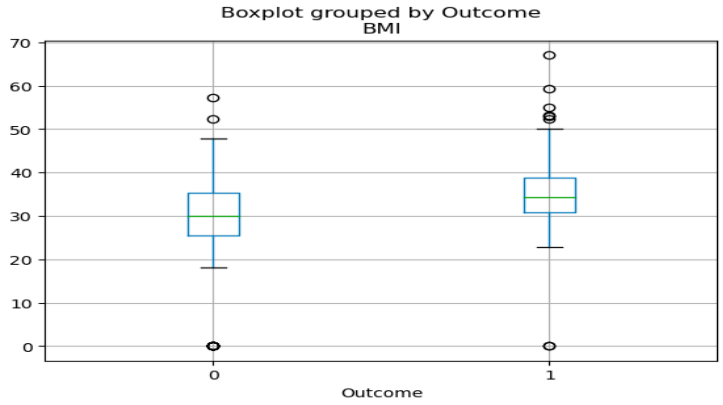
```
df['Outcome'].value_counts().plot.bar()
```



Box plots in pandas

The quartile distribution of continuous variables can be visualized using a boxplot. The code below lets you create a boxplot with pandas.

```
df.boxplot(column=['BMI'], by='Outcome')
```



Conclusion:

Pandas is a fundamental tool for data analysis in Python. Its powerful data structures, intuitive syntax, and comprehensive functionality make it essential for anyone working with data, whether for data cleaning, analysis, or visualization.

Pandas vs Seaborn

Feature	Pandas	Seaborn
Primary Purpose	Data manipulation and analysis	Statistical data visualization
Core Data Structures	Series (1D) and DataFrame (2D)	Utilizes DataFrames for plotting
Data Manipulation	Extensive functions for cleaning, transforming, filtering, and aggregating data	Minimal; relies on Pandas for preprocessing
Visualization Capabilities	Basic plotting with `DataFrame.plot()`	Advanced visualization functions (e.g., `sns.boxplot()`, `sns.scatterplot()`)
Customization Options	Limited customization via Matplotlib	Extensive options for themes, color palettes, and styles
Statistical Features	Basic statistics, not focused on visualizing statistical relationships	Built-in statistical visualizations (e.g., confidence intervals, regression lines)
Ease of Use	User-friendly for data manipulation, but plotting can require additional effort	High-level API simplifies complex visualizations
Dependency	Standalone library	Built on top of Matplotlib
Visual Aesthetics	Functional but less visually appealing	Aesthetically pleasing defaults and themes

Intermediate task:

Conduct an in-depth analysis of the Air Quality Index (AQI) in Delhi, addressing the specific environmental challenges faced by the city. Define research questions centered around key pollutants, seasonal variations, and the impact of geographical factors on air quality. Utilize statistical analyses and visualizations to gain insights into the dynamics of AQI in Delhi, offering a comprehensive understanding that can inform targeted strategies for air quality improvement and public health initiatives in the region.

Introduction to the Air Quality Index Report: Delhi

Delhi, one of the most densely populated and industrialized cities in the world, faces significant challenges related to air pollution. The Air Quality Index (AQI) serves as a crucial tool for evaluating and communicating the state of air quality in the region. This report provides an in-depth analysis of the AQI in Delhi, derived from a dataset that includes key pollutants: carbon monoxide (CO), nitrogen oxides (NO and NO2), ozone (O3), sulfur dioxide (SO2), particulate matter (PM2.5 and PM10), and ammonia (NH3).

The dataset encompasses air quality measurements collected from various monitoring stations across the city over the past year. These pollutants have diverse sources, including vehicular emissions, industrial discharges, construction activities, and seasonal factors like crop burning, all of which significantly influence air quality levels.

In this report, we will analyze trends in pollutant concentrations and their correlation with AQI values, highlighting critical periods of poor air quality and their implications for public health. We will also discuss the potential health risks associated with different pollutant levels, particularly for vulnerable populations.

Air Quality Index Analysis: Process We Can Follow

Air Quality Index Analysis aims to provide a numerical value representative of overall air quality, essential for public health and environmental management. Below are the steps we can follow for the task of Air Quality Index Analysis:

1. Gather air quality data from various sources, such as government monitoring stations, sensors, or satellite imagery.
2. Clean and preprocess the collected data.
3. Calculate the Air Quality Index using standardized formulas and guidelines provided by environmental agencies.
4. Create visualizations, such as line charts or heatmaps, to represent the AQI over time or across geographical regions.
5. Compare the AQI metrics of the location with the recommended air quality metrics.

Air Quality Index Analysis using Python

Now, let's get started with the task of Air Quality Index Analysis by importing the necessary Python libraries and the **dataset**:

Code:

```
import pandas as pd
import numpy as np
from google.colab import files
uploaded=files.upload()
data = pd.read_csv("delhiaqi.csv")
print(data.head())
```

Output:

```
                 date       co     no    no2    o3   so2    pm2_5   pm10  \
0 2023-01-01 00:00:00  1655.58   1.66  39.41  5.90  17.88  169.29  194.64
1 2023-01-01 01:00:00  1869.20   6.82  42.16  1.99  22.17  182.84  211.08
2 2023-01-01 02:00:00  2510.07  27.72  43.87  0.02  30.04  220.25  260.68
3 2023-01-01 03:00:00  3150.94  55.43  44.55  0.85  35.76  252.90  304.12
4 2023-01-01 04:00:00  3471.37  68.84  45.24  5.45  39.10  266.36  322.80

    nh3
0  5.83
1  7.66
2 11.40
3 13.55
4 14.19
```

The date column in the dataset into a datetime data type and move forward:

```
data['date'] = pd.to_datetime(data['date'])
```

Now, let's have a look at the descriptive statistics of the data:

Code:

```
print(data.describe())
```

Output:

```
                co           no          no2          o3          so2  \
count   561.000000   561.000000   561.000000   561.000000   561.000000
mean   3814.942210    51.181979    75.292496    30.141943    64.655936
std    3227.744681    83.904476    42.473791    39.979405    61.073080
min     654.220000     0.000000    13.370000     0.000000     5.250000
25%    1708.980000     3.380000    44.550000     0.070000    28.130000
50%    2590.180000    13.300000    63.750000    11.800000    47.210000
75%    4432.680000    59.010000    97.330000    47.210000    77.250000
max   16876.220000   425.580000   263.210000   164.510000   511.170000
```

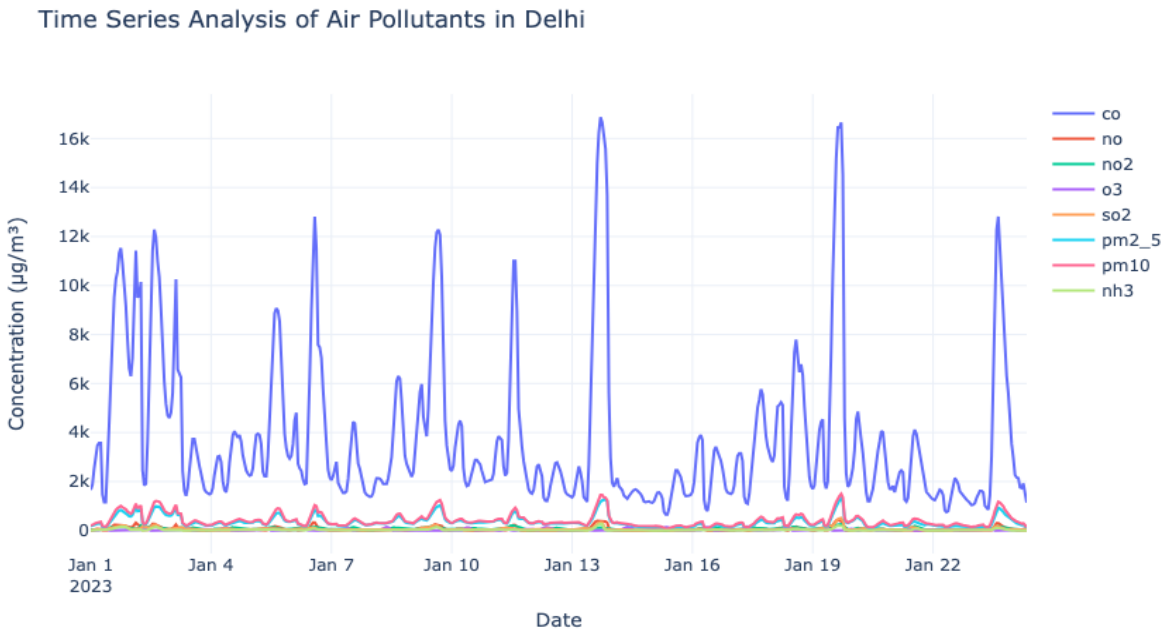
	pm2_5	pm10	nh3
count	561.000000	561.000000	561.000000
mean	358.256364	420.988414	26.425062
std	227.359117	271.287026	36.563094
min	60.100000	69.080000	0.630000
25%	204.450000	240.900000	8.230000
50%	301.170000	340.900000	14.820000
75%	416.650000	482.570000	26.350000
max	1310.200000	1499.270000	267.510000

Now let's have a look at the intensity of each pollutant over time in the air quality:

Code:

```
# time series plot for each air pollutant
fig = go.Figure()

for pollutant in ['co', 'no', 'no2', 'o3', 'so2', 'pm2_5', 'pm10', 'nh3']:

    fig.add_trace(go.Scatter(x=data['date'], y=data[pollutant], mode='lines',

                             name=pollutant))

fig.update_layout(title='Time Series Analysis of Air Pollutants in Delhi',

                  xaxis_title='Date', yaxis_title='Concentration (µg/m³)')
fig.show()
```

Output:



In the above code, we are creating a time series plot for each air pollutant in the dataset. It helps analyze the intensity of air pollutants over time.

Calculating Air Quality Index:

Now, before moving forward, we need to calculate the air quality index and its category. AQI is typically computed based on the concentration of various pollutants, and each pollutant has its sub-index.

Here's how we can calculate AQI:

Code:

```
# Define AQI breakpoints and corresponding AQI values
aqi_breakpoints = [
    (0, 12.0, 50), (12.1, 35.4, 100), (35.5, 55.4, 150),
    (55.5, 150.4, 200), (150.5, 250.4, 300), (250.5, 350.4, 400),
    (350.5, 500.4, 500)
]
def calculate_aqi(pollutant_name, concentration):
    for low, high, aqi in aqi_breakpoints:
        if low <= concentration <= high:
            return aqi
    return None

def calculate_overall_aqi(row):
    aqi_values = []
    pollutants = ['co', 'no', 'no2', 'o3', 'so2', 'pm2_5', 'pm10', 'nh3']
    for pollutant in pollutants:
        aqi = calculate_aqi(pollutant, row[pollutant])
        if aqi is not None:
            aqi_values.append(aqi)
    return max(aqi_values)
# Calculate AQI for each row
data['AQI'] = data.apply(calculate_overall_aqi, axis=1)

# Define AQI categories
aqi_categories = [
    (0, 50, 'Good'), (51, 100, 'Moderate'), (101, 150, 'Unhealthy for Sensitive Groups'),
    (151, 200, 'Unhealthy'), (201, 300, 'Very Unhealthy'), (301, 500, 'Hazardous')
]

def categorize_aqi(aqi_value):
    for low, high, category in aqi_categories:
        if low <= aqi_value <= high:
            return category
    return None
```

```
# Categorize AQI
data['AQI Category'] = data['AQI'].apply(categorize_aqi)
print(data.head())
```

Output:

```
                 date       co    no    no2    o3    so2   pm2_5    pm10  \
0 2023-01-01 00:00:00  1655.58  1.66  39.41  5.90  17.88  169.29  194.64
1 2023-01-01 01:00:00  1869.20  6.82  42.16  1.99  22.17  182.84  211.08
2 2023-01-01 02:00:00  2510.07 27.72  43.87  0.02  30.04  220.25  260.68
3 2023-01-01 03:00:00  3150.94 55.43  44.55  0.85  35.76  252.90  304.12
4 2023-01-01 04:00:00  3471.37 68.84  45.24  5.45  39.10  266.36  322.80

    nh3  AQI     AQI Category
0  5.83  300  Very Unhealthy
1  7.66  300  Very Unhealthy
2 11.40  400        Hazardous
3 13.55  400        Hazardous
4 14.19  400        Hazardous
```

In the above code, we are defining AQI breakpoints and corresponding AQI values for various air pollutants according to the Air Quality Index (AQI) standards. The aqi_breakpoints list defines the concentration ranges and their corresponding AQI values for different pollutants. We then define two functions:

1. **Calculate AQI:** to calculate the AQI for a specific pollutant and concentration by finding the appropriate range in the aqi_breakpoints.

2. **Calculate_overall_aqi:** to calculate the overall AQI for a row in the dataset by considering the maximum AQI value among all pollutants.

The calculated AQI values are added as a new column in the dataset. Additionally, we defined AQI categories in the aqi_categories list and used the categorize_aqi function to assign an AQI category to each AQI value. The resulting AQI categories are added as a new column as AQI Category in the dataset.

Analyzing AQI of Delhi:

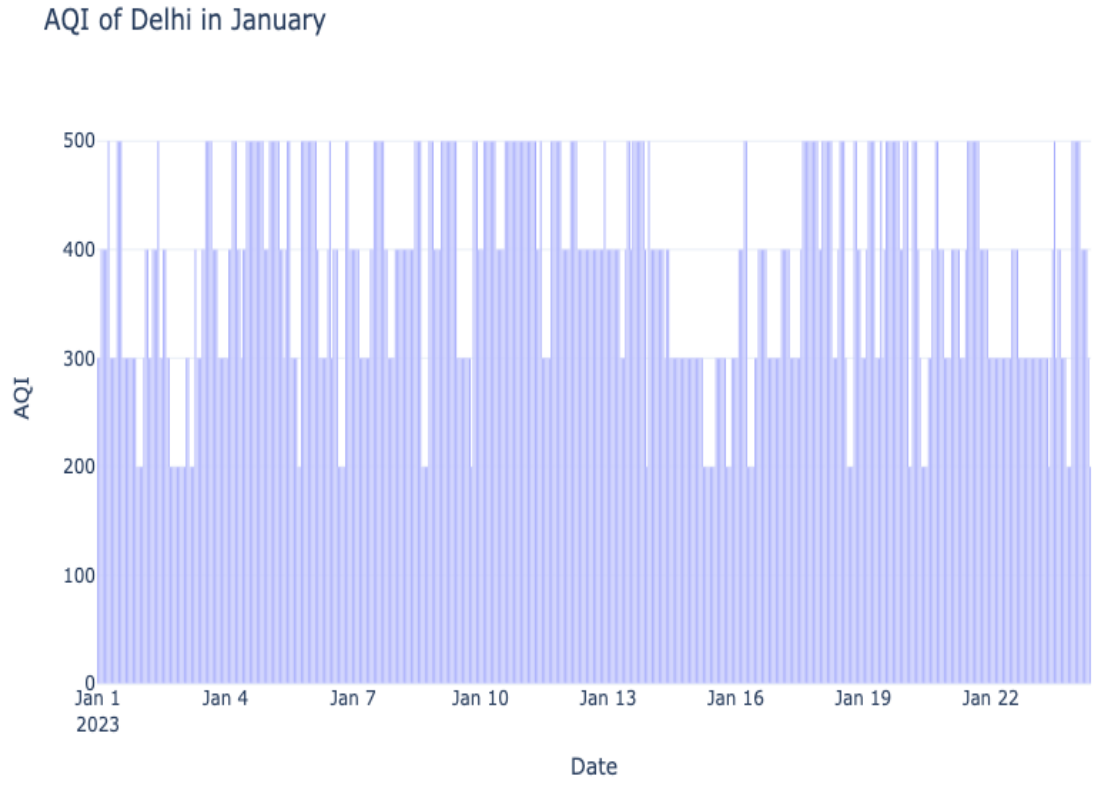
Now, let's have a look at the AQI of Delhi in January:

Code:

```
# AQI over time
fig = px.bar(data, x="date", y="AQI",
             title="AQI of Delhi in January")
```

```
fig.update_xaxes(title="Date")
fig.update_yaxes(title="AQI")
fig.show()
```

Output:

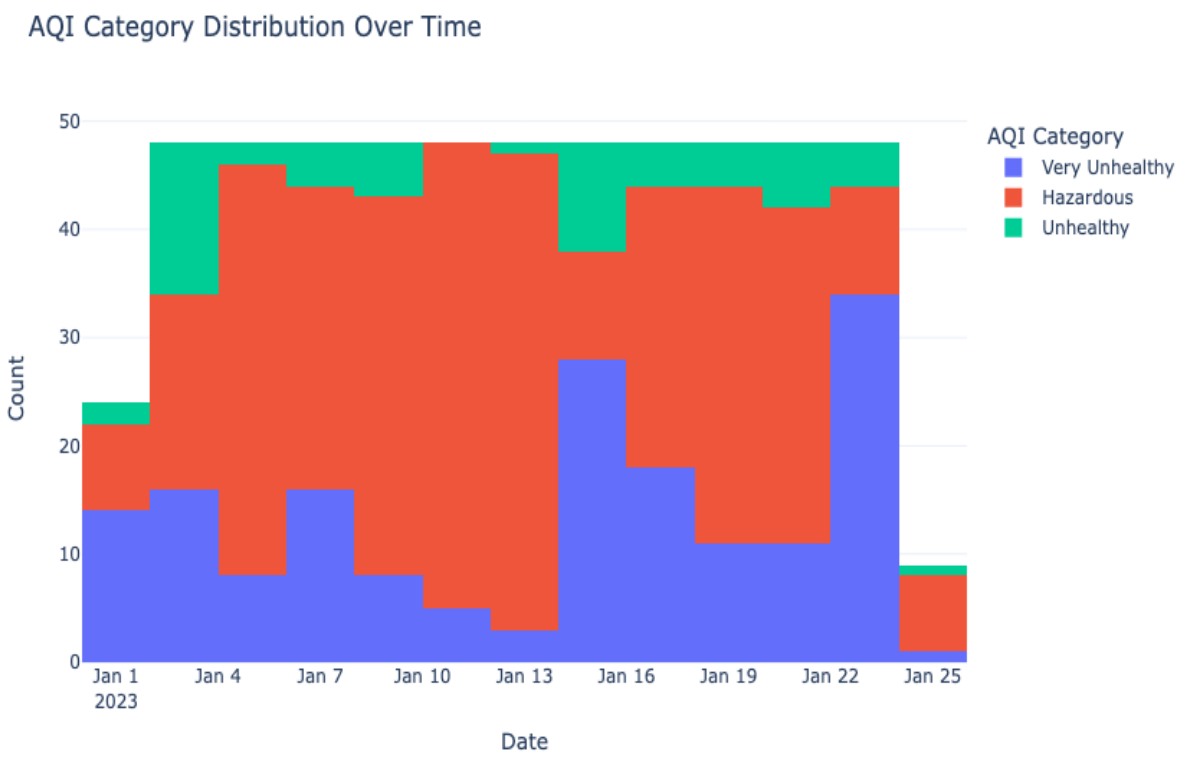


Now, let's have a look at the AQI category distribution:

Code:

```
fig = px.histogram(data, x="date",
                   color="AQI Category",
                   title="AQI Category Distribution Over Time")
fig.update_xaxes(title="Date")
fig.update_yaxes(title="Count")
fig.show()
```

Output:



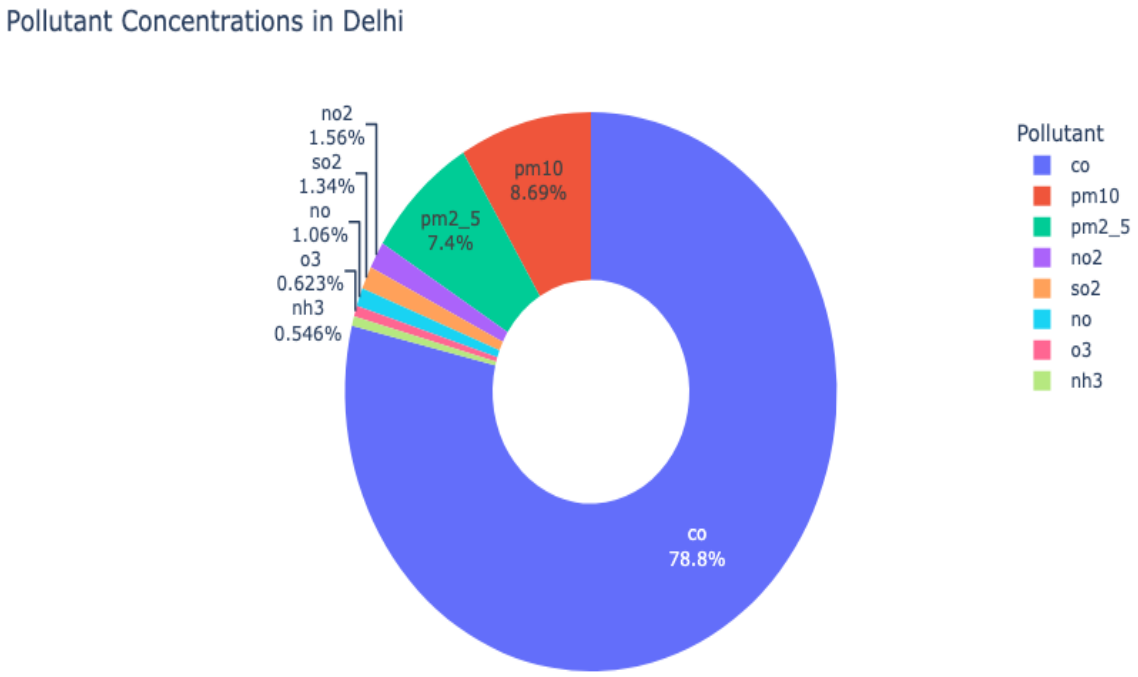
Now, let's have a look at the distribution of pollutants in the air quality of Delhi:

Code:

```
# Define pollutants and their colors
pollutants = ["co", "no", "no2", "o3", "so2", "pm2_5", "pm10", "nh3"]
pollutant_colors = px.colors.qualitative.Plotly

# Calculate the sum of pollutant concentrations
total_concentrations = data[pollutants].sum()

# Create a DataFrame for the concentrations
concentration_data = pd.DataFrame({
    "Pollutant": pollutants,
    "Concentration": total_concentrations
```

```
})

# Create a donut plot for pollutant concentrations
fig = px.pie(concentration_data, names="Pollutant", values="Concentration",
             title="Pollutant Concentrations in Delhi",
             hole=0.4, color_discrete_sequence=pollutant_colors)

# Update layout for the donut plot
fig.update_traces(textinfo="percent+label")
fig.update_layout(legend_title="Pollutant")

# Show the donut plot
fig.show()
```

Output:

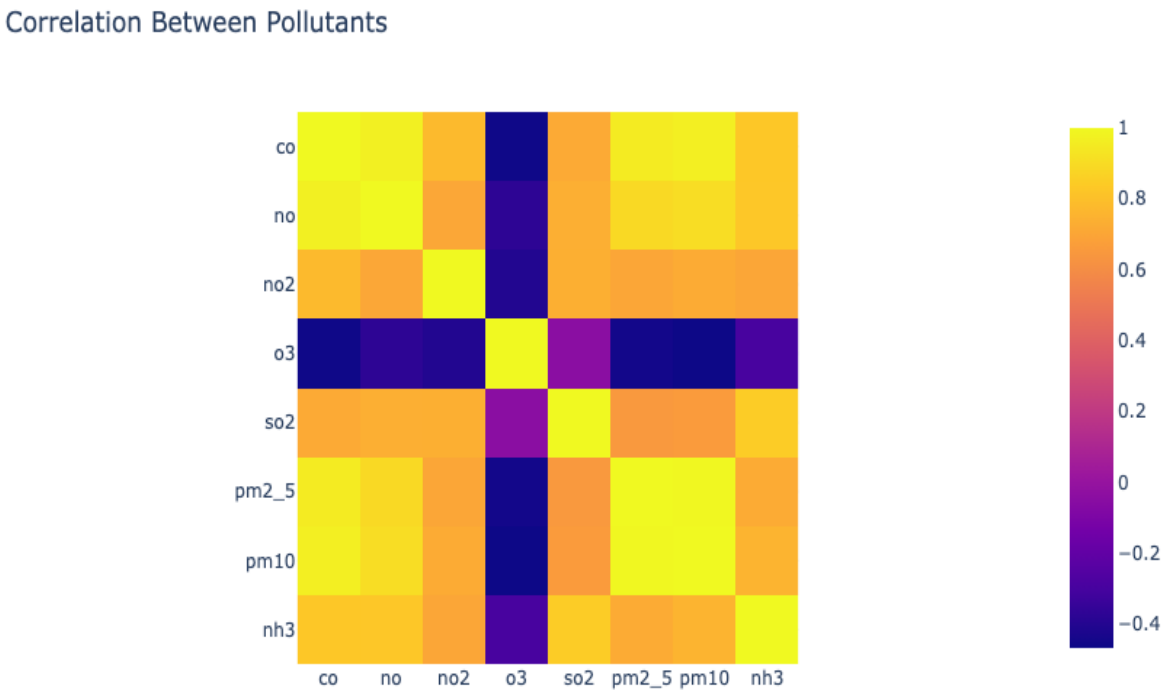


Now, let's have a look at the correlation between pollutants:

Code:

```
# Correlation Between Pollutants
correlation_matrix = data[pollutants].corr()
fig = px.imshow(correlation_matrix, x=pollutants,
                y=pollutants, title="Correlation Between Pollutants")
fig.show()
```

Output:



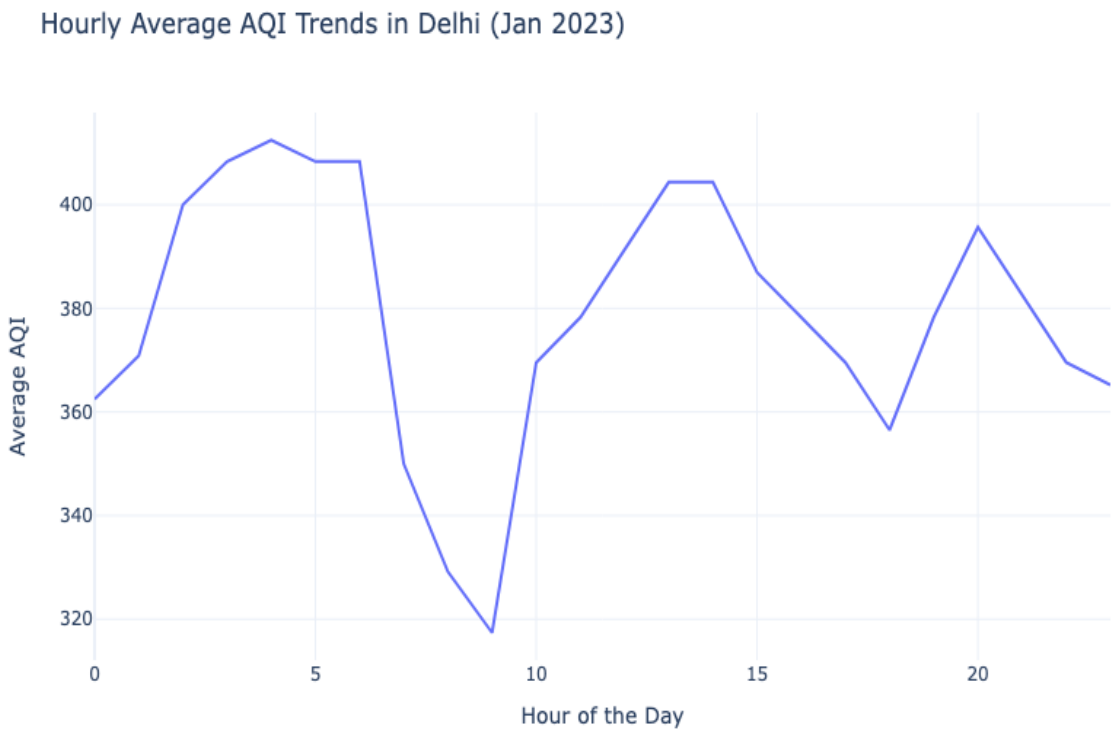
The correlation matrix displayed here represents the correlation coefficients between different air pollutants in the dataset. Correlation coefficients measure the strength and direction of the linear relationship between two variables, with values ranging from -1 to 1. Overall, the positive correlations among CO, NO, NO2, SO2, PM2.5, PM10, and NH3 suggest that they may share common sources or have similar pollution patterns, while O3 exhibits an inverse relationship with the other pollutants, which may be due to its role as both a pollutant and a natural atmospheric oxidant.

Now, let's have a look at the hourly average trends of AQI in Delhi:

Code:

```
# Extract the hour from the date
data['Hour'] = pd.to_datetime(data['date']).dt.hour
# Calculate hourly average AQI
hourly_avg_aqi = data.groupby('Hour')['AQI'].mean().reset_index()
# Create a line plot for hourly trends in AQI
fig = px.line(hourly_avg_aqi, x='Hour', y='AQI',
        title='Hourly Average AQI Trends in Delhi (Jan 2023)')
fig.update_xaxes(title="Hour of the Day")
fig.update_yaxes(title="Average AQI")
fig.show()
```

Output:

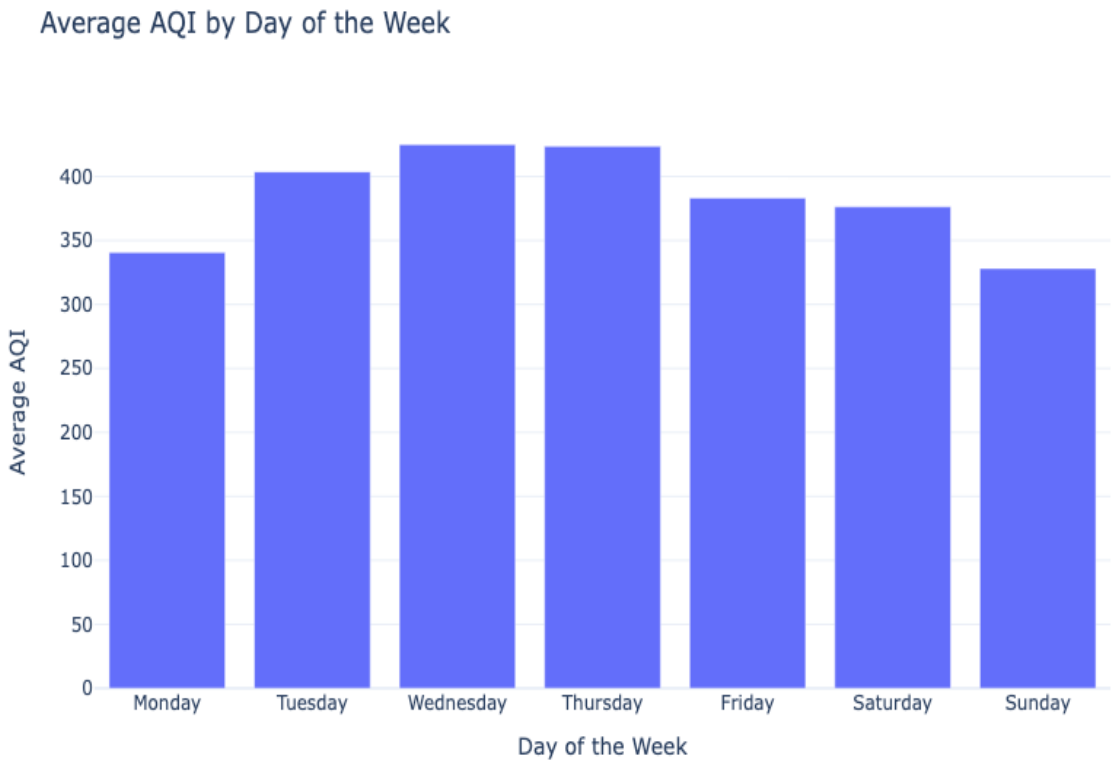


Now, let’s have a look at the average AQI by day of the week in Delhi:

Code:

```
# Average AQI by Day of the Week
data['Day_of_Week'] = data['date'].dt.day_name()
average_aqi_by_day = data.groupby('Day_of_Week')['AQI'].mean().reindex(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'])
fig = px.bar(average_aqi_by_day, x=average_aqi_by_day.index, y='AQI',
             title='Average AQI by Day of the Week')
fig.update_xaxes(title="Day of the Week")
fig.update_yaxes(title="Average AQI")
fig.show()
```

Output:



It shows that the air quality in Delhi is worse on Wednesdays and Thursdays.

Summary:

Air quality index (AQI) analysis is a crucial aspect of environmental data science that involves monitoring and analyzing air quality in a specific location. It aims to provide a numerical value representative of overall air quality, essential for public health and environmental management.