upGrad

SUMMARY

Introduction to Data Visualisation

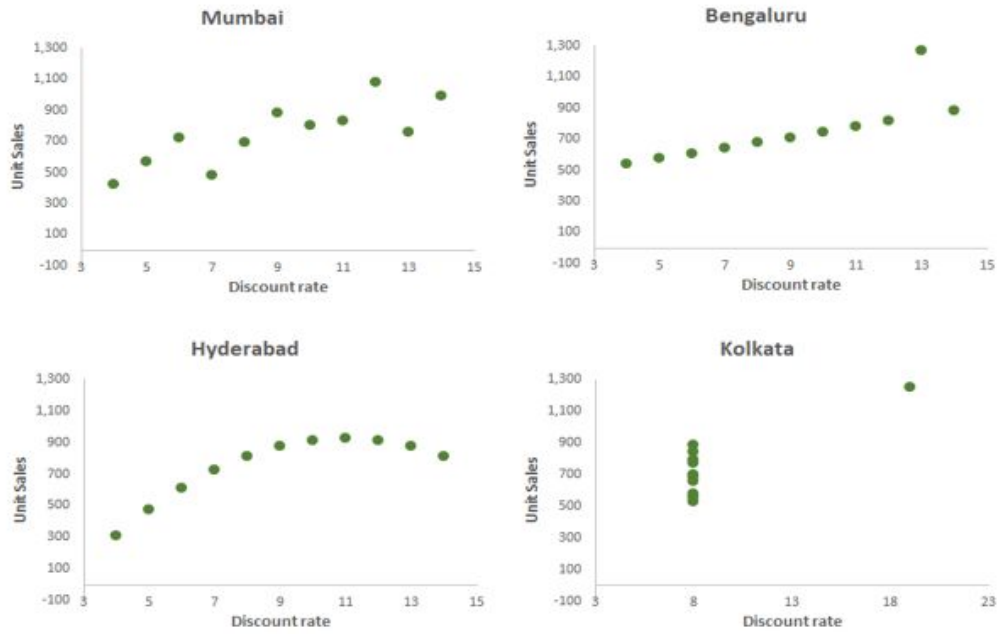
In this module, you will learn about the importance of data visualisation in the real world, data handling and cleaning, sanity checks, and various charts and plots, all of which can be used to observe trends, obtain relationships, and portray final results.

The Necessity of Data Visualisation

The major advantage of data visualisation is that we can decipher the underlying patterns from the raw numbers which are otherwise difficult to see for the human eye. Therefore, it is important to visualise the data to observe how different features behave. The following example shows the distribution of sales corresponding to specific discount rates in four major cities. By observing the raw numbers, it is difficult to conclude anything because both the statistics - average and standard deviation - are equal across cities. However, when the relationship between sales and discount rate is obtained, it is observed that different cities follow their own trends.

	Mumbai		Bengaluru		Hyderabad		Kolkata	
Month	Discount	Sales	Discount	Sales	Discount	Sales	Discount	Sales
January	10	804	10	914	10	746	8	658
February	8	695	8	814	8	677	8	576
March	13	758	13	874	13	1,274	8	771
April	9	881	9	877	9	711	8	884
May	11	833	11	926	11	781	8	847
June	14	996	14	810	14	884	8	704
July	6	724	6	613	6	608	8	525
August	4	426	4	310	4	539	19	1,250
September	12	1,084	12	913	12	815	8	556
October	7	482	7	726	7	642	8	791
November	5	568	5	474	5	574	8	689
Average	**9**	**750.1**	**9**	**750.1**	**9**	**750.1**	**9**	**750.1**
Std. Dev.	**3.16**	**193.7**	**3.16**	**193.7**	**3.16**	**193.7**	**3.16**	**193.7**

However, the patterns in the underlying data and the difference become apparent when visualised through appropriate plots.



Anscombe's Quartet

Here, the trend followed by each city is different.

- Mumbai: A straight line with little variance.
- Bengaluru: A straight line with a sudden rise in unit sales at a 13% discount rate.
- Hyderabad: A curvy line with maximum unit sales at 11% followed by a decreasing trend.
- Kolkata: As the discount rate is fixed for almost all months except August, unit sales are in the range of 500-900.

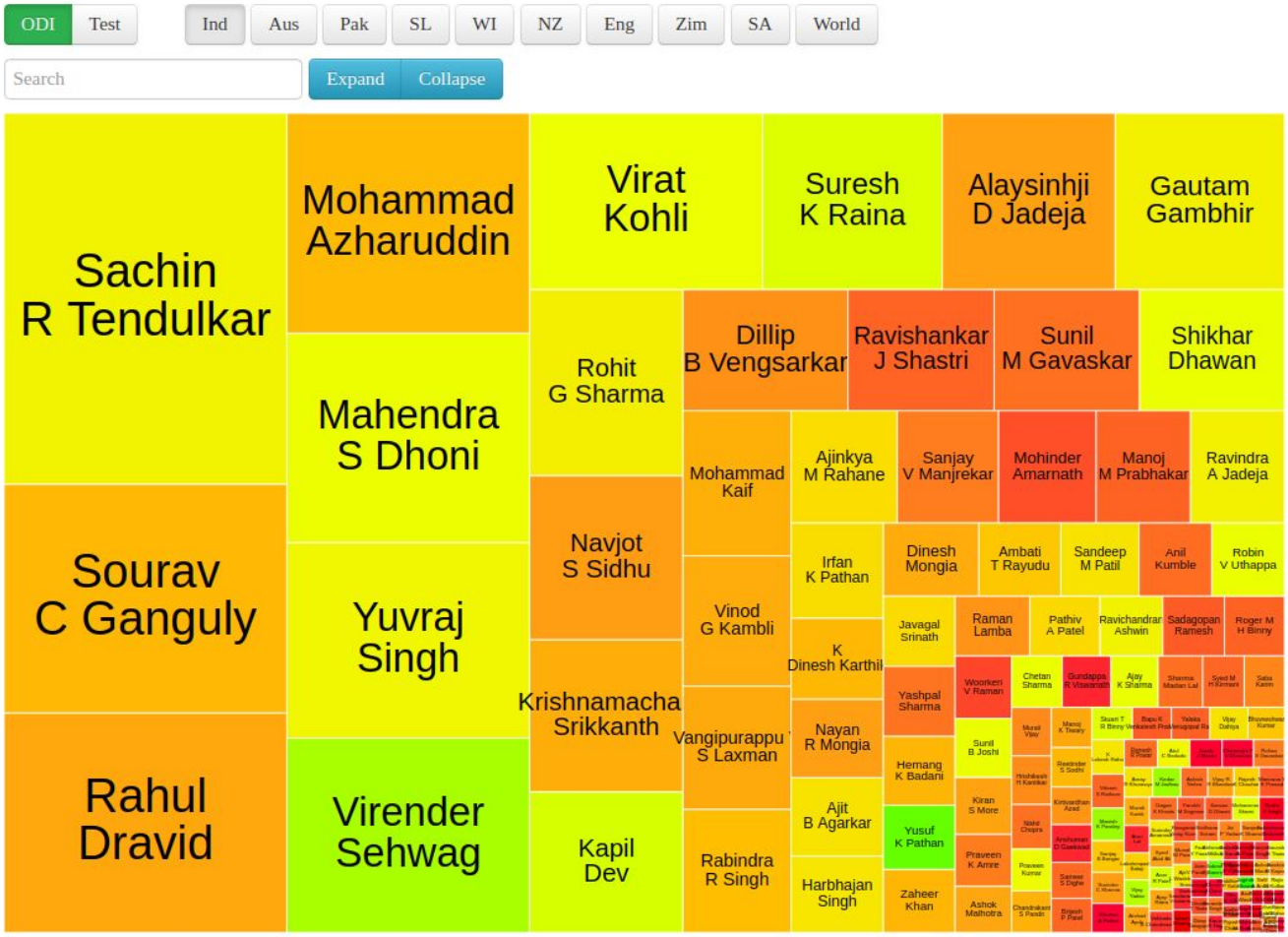
Each branch employs different strategies to calculate its discount rates. Moreover, sales numbers were also different across branches. Therefore, one should utilise an appropriate visualisation technique to 'look' into the data.

upGrad

Visualisation - Some Examples

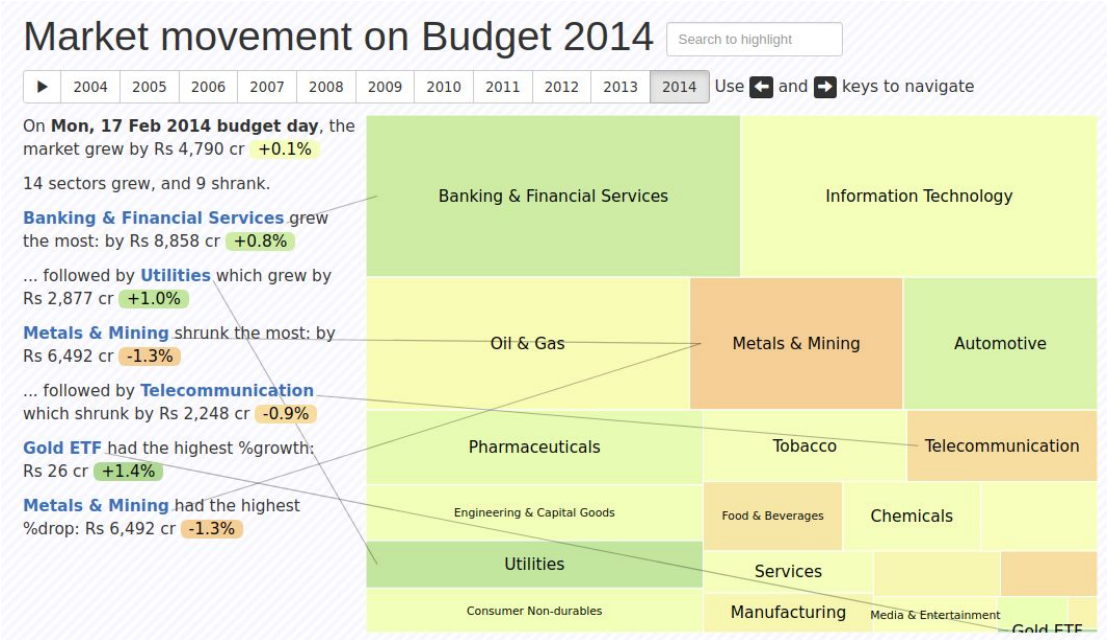
1. For the popular cricket-playing nations, the following visualisation illustrates every batsman who has scored at least 20 runs in their ODI career along with their run rate. Size = Number of runs.

Colour = Strike rate 20 50 80 110 140. You can play with the visualisation here.



Each block is further divided into several blocks representing the runs scored by the cricketers in each innings.

2. A treemap diagram showing how multiple companies and sectors react to the budget. You can find the interactive graph here.



3. Visual exploratory analytics: Data visualisation helps in understanding the connections between different software and clustering them based on common features. You can find the visual here.

Data Cleaning and Handling

Once you load a dataset, there is a possibility of many disturbances being present. The most common ones are missing values and incorrect data types. Following are some common techniques to address these issues:

For missing values: Following are some common techniques to address this issue:

- Dropping the rows containing the missing values

upGrad

Example:

```
#Check the number of null values in the columns
inp0.isnull().sum()
```

```
#Drop the rows having null values in the Rating field
inp1 = inp0[~inp0.Rating.isnull()]

#Check the shape of the dataframe
inp1.shape
```

- Imputing the missing values
  - For numerical variables, use mean and median
  - For categorical variables, use mode

Example:

```
#Check the most common value in the Android version column
inp1['Android Ver'].value_counts()
```

```
#Replace the nulls in the Current version column with the above value
inp1['Current Ver'] = inp1['Current Ver'].fillna(inp1['Current Ver'].mode()[0])
```

- Keep the missing values if they do not affect the analysis

Incorrect data types:

- Clean certain values
- Clean and convert an entire column

Example:

```
#Write the function to make the changes
inp1.Price = inp1.Price.apply(lambda x: 0 if x=="0" else float(x[1:]))
```

```
#Change the dtype of this column
inp1.Reviews = inp1.Reviews.astype("int32")
```

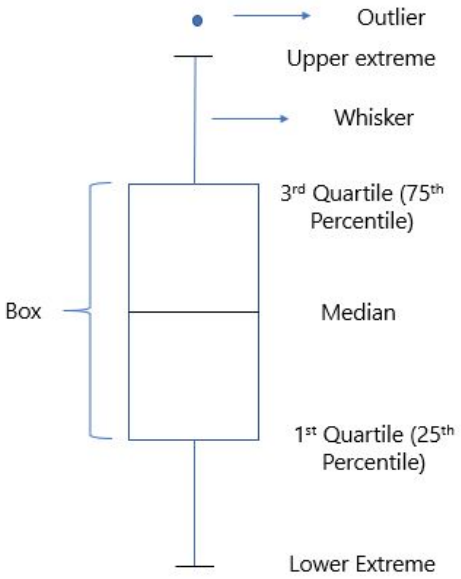
upGrad

Sanity Checks

Often, our data is noisy, incomplete, irregular or dirty. Before coming to any conclusions from those beautiful charts, we need to perform sanity checks on the data available to ensure nothing is wrong. These sanity checks are part of the process of data preparation and cleaning it forms a good chunk of the data scientist's time. For example, one should ensure that certain features such as price, weight, height, etc., are always positive in the column. In our Google Play app case study, we ensured that the number of reviews is always less than or equal to the number of installs. We can always remove all the rows which do not make any sense at all.

Outliers Analysis with Box Plot

Outliers are extreme values that deviate from other observations on **data**. They may indicate variability in measurement, experimental errors, or a novelty. In other words, an **outlier** is an observation that diverges from an overall pattern on a sample. This is where one should start utilising visualisation to achieve tasks. The visualisation best suited for this is the **box plot**.



The maximum and minimum values are represented by the fences of the box plot. The maximum value is given by the formula **Q3+1.5*IQR**, and the minimum value is represented by the formula **Q1-1.5*IQR.**

IQR: Interquartile range that denotes the values lying between the percentiles 25 and 75.

Outliers are the values outside the range mentioned earlier.

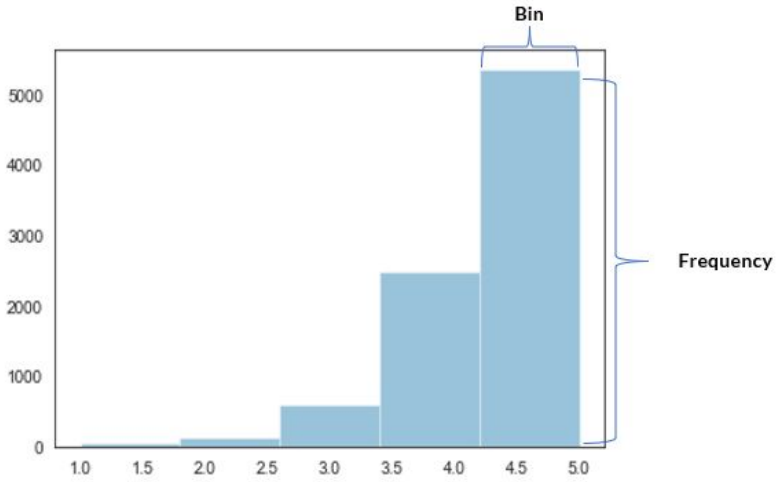
```
#Create a box plot for the price column
plt.boxplot(inp1.Price)
plt.show()
```

```
#Create a box plot for paid apps
inp1[inp1.Price>0].Price.plot.box()
```

```
#Clean the Price column again
inp1 = inp1[inp1.Price <= 30]
inp1.shape
```

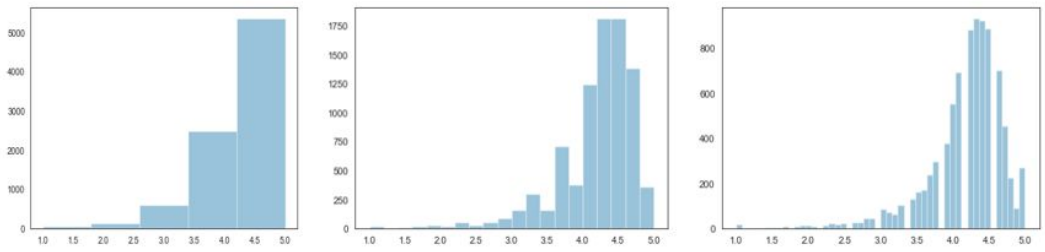
In the previous example, all outlier points greater than 30 were removed from the dataset.

Histogram



Histograms generally work by bucketing the entire range of values that a particular variable takes to specific bins.

Vertical bars denote the total number of records in a specific bin, which is also known as its **frequency**. The number of bins can be increased or decreased to change the granularity of the analysis.



Code for plotting a histogram:

```
plt.hist(inp1.Reviews)
plt.show()
```

upGrad

SUMMARY

Data Visualisation with Seaborn

Seaborn:

- Python library to create statistical graphs easily
- Built on top of matplotlib and closely integrated with *pandas*

Functionalities of Seaborn:

- Dataset-oriented API
- Analysing univariate and bivariate distributions
- Automatic estimation and plotting of linear regression models
- Convenient views for complex datasets
- Concise control over the style
- Colour palettes

Importing Seaborn:

```
#import the necessary libraries
import warnings
warnings.filterwarnings("ignore")
import seaborn as sns
```

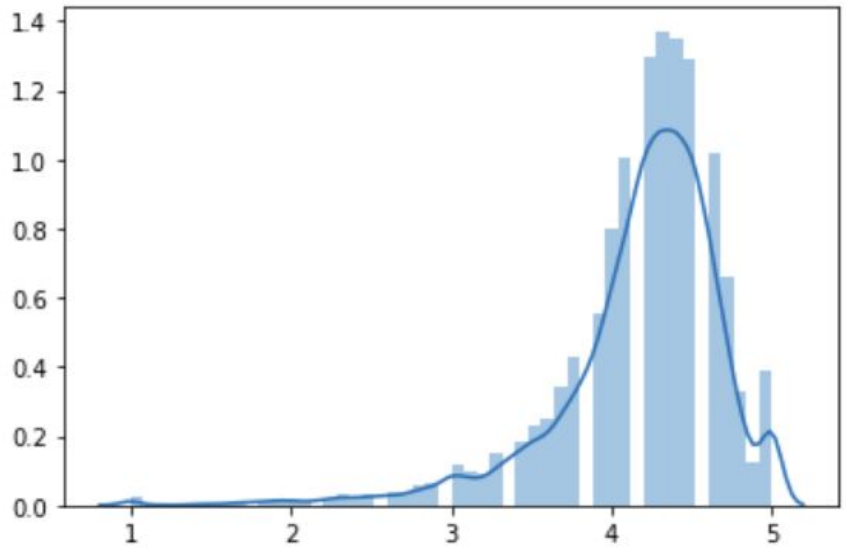
Distribution Plots

A distribution plot is pretty similar to the histogram functionality in matplotlib. Instead of a frequency plot, it plots an approximate probability density for that rating bucket. The curve or the **KDE** that gets drawn over the distribution is the approximate probability density curve.

Following is an example of a distribution plot. Notice that the left axis has the density for each bin or bucket instead of the frequency.

Code:

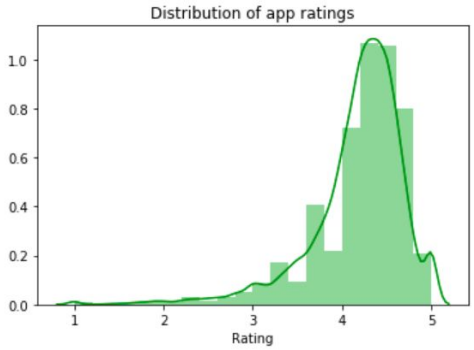
```
#Create a distribution plot for rating
sns.distplot(inp1.Rating)
plt.show()
```



Distribution Plot

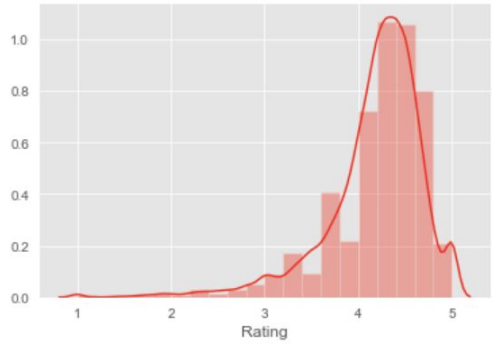
Some styling options in distribution plots:

```
#Apply matplotlib functionalities
sns.distplot(inp1.Rating, bins=20, color="g")
plt.title("Distribution of app ratings", fontsize=12)
plt.show()
```




```
plt.style.use("ggplot")
```

```
sns.distplot(inp1.Rating, bins=20)
plt.show()
```

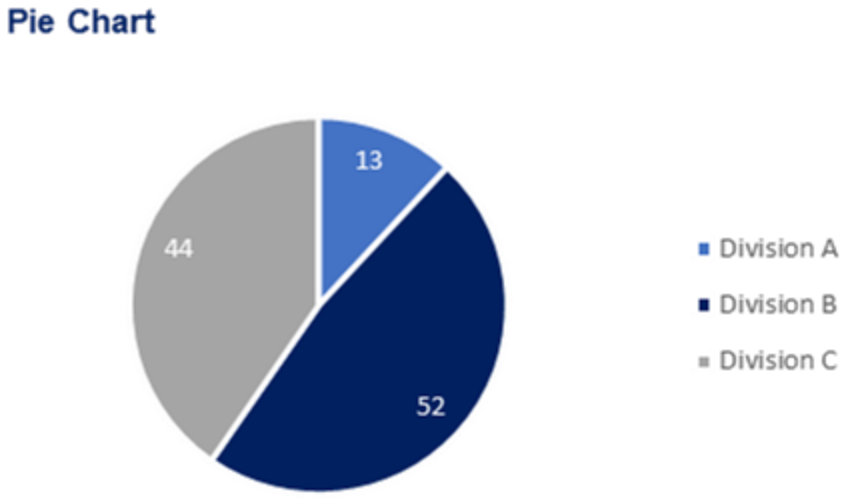


upGrad

Pie Chart and Bar Chart

To analyse categorical columns, pie charts and bar charts are used to portray the relationships.

1. **Pie Chart:** A pie chart is a circular graph divided into slices. The larger a slice is, the bigger portion of the total quantity it represents. For example, if a company operates three separate divisions, at the year end, its top management would be interested in seeing what portion of total revenue each division accounted for.



2. **Bar Chart:** Bar charts are among the most frequently used chart types. As the name suggests, a bar chart is composed of a series of bars illustrating a variable's development. Given that bar charts are such a common chart type, people are generally familiar with them and can understand them easily. Examples like the following one are straightforward to read.

Code:

```
#Plot a pie chart
inp1['Content Rating'].value_counts().plot.pie()
plt.show()
```

```
#Plot a bar chart
inp1['Content Rating'].value_counts().plot.bar()
plt.show()
```

To plot a horizontal bar chart:

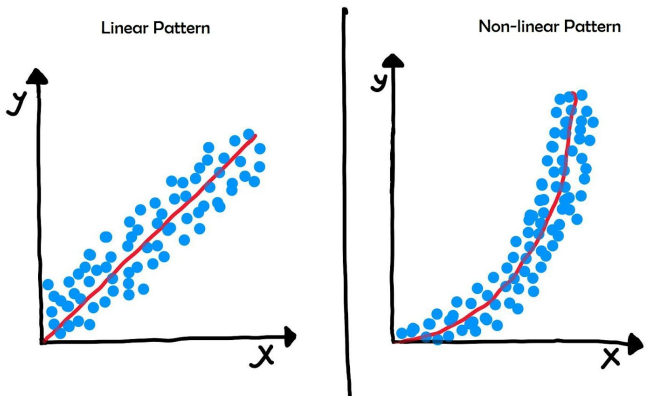
```
inp1['Content Rating'].value_counts().plot.barh()
plt.show()
```

Scatter Plot

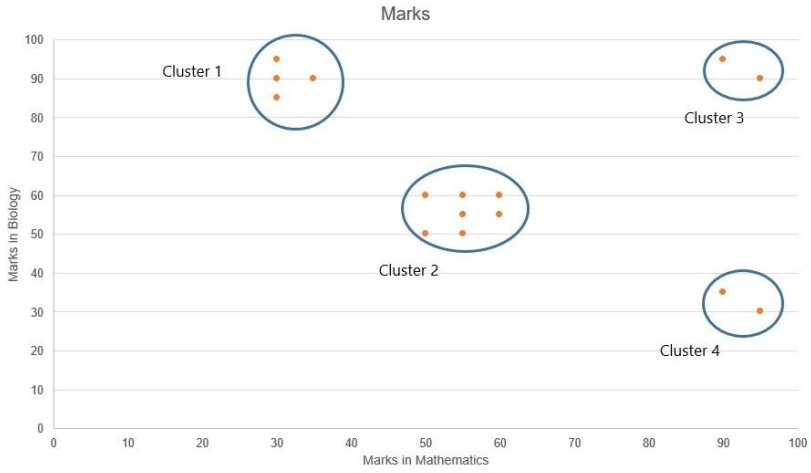
Scatter plots are perhaps one of the most commonly used and powerful visualisations used in the field of machine learning. They are crucial in revealing relationships between the data points. And one can generally deduce some sort of trends in the data with the help of a scatter plot.

Applications of Scatter Plots in Machine Learning:

- Scatter plots are useful in regression problems to check whether a linear trend exists in the data or not. For example, in the following image, creating a linear model in the first case makes far more sense as a clear straight-line trend is visible.



- Scatter plots help in observing naturally occurring clusters. In the following image, the marks of students in Maths and Biology has been plotted. You can clearly group the students into four clusters. Cluster one includes students who score very well in Biology but very poorly in Maths, Cluster two has students who score equally well in both the subjects, and so on.

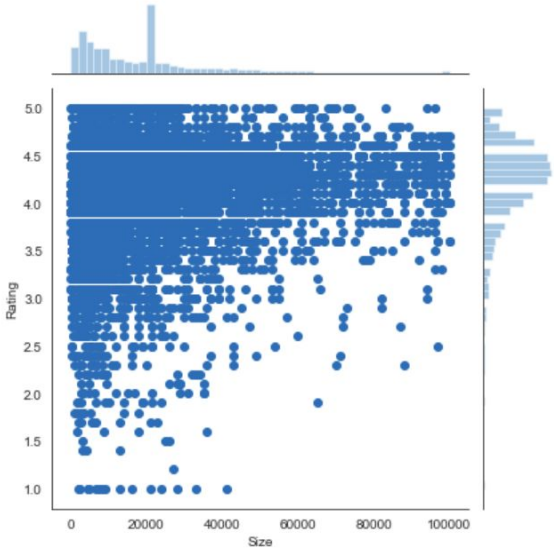


Code:

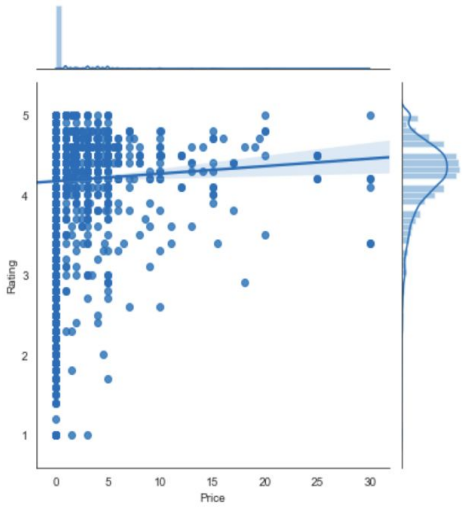
```
###Size vs Rating

##Plot a scatter-plot in the matplotlib way between Size and Rating
plt.scatter(inp1.Size, inp1.Rating)
plt.show()
```

- **Joint plot** displays a relationship between two variables. On the other hand, **Reg plots** are an extension to the joint plots with the addition of a regression line to the view.

```
sns.jointplot(inp1.Size, inp1.Rating)
plt.show()
```




```
##Plot a reg plot for Price and Rating and observe the trend
sns.jointplot(inp1.Price, inp1.Rating, kind="reg")
plt.show()
```

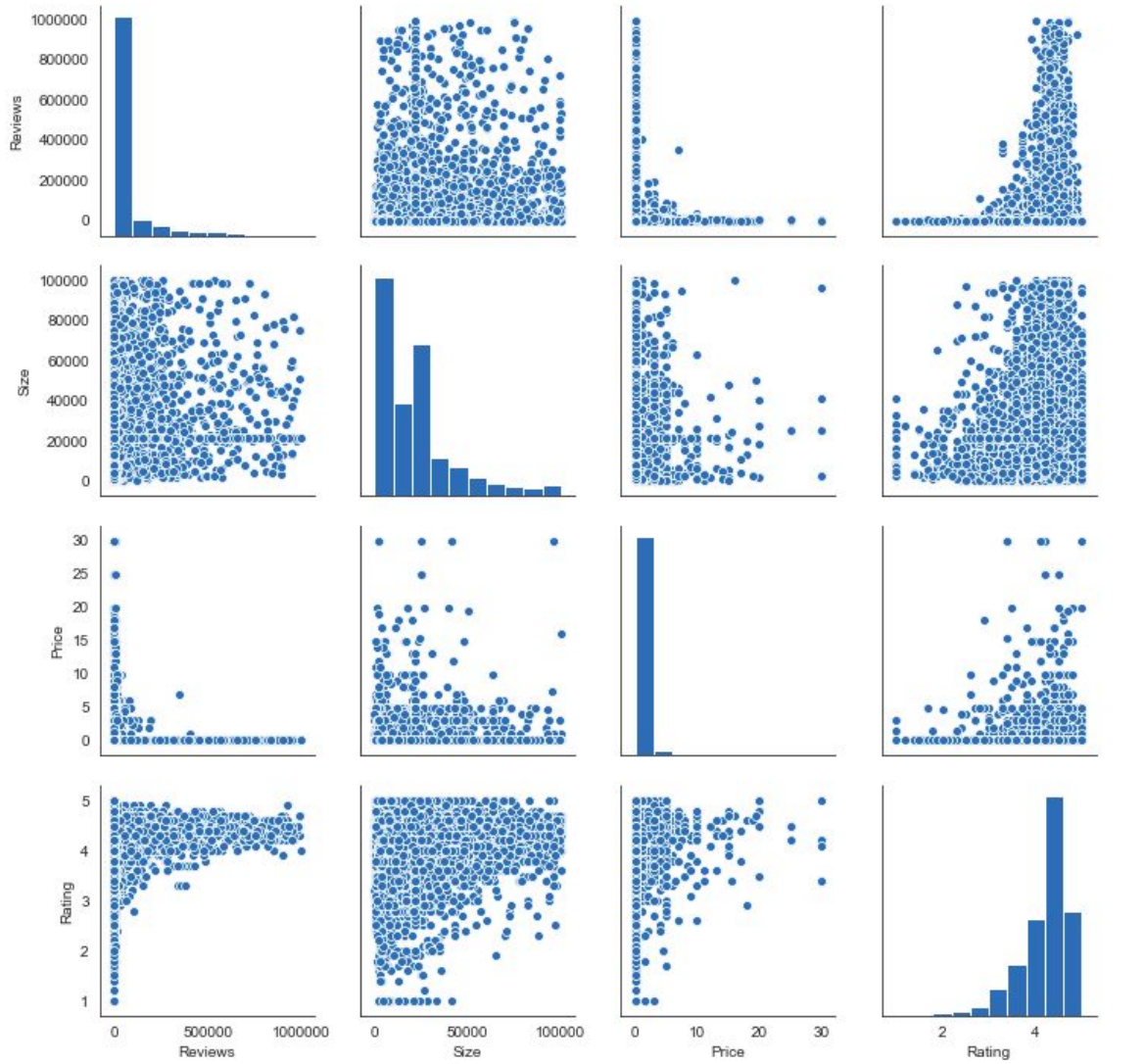


upGrad

Pair Plots

Pair plots help in identifying quickly the trends between a target variable and the predictor variables. For example, suppose you want to predict how your company's profits are affected by three different factors. In order to choose one factor, you created a pair plot containing profits and the three factors as variables. Here are the scatter plots of profits vs the three variables that you obtained from the pair plot.

```
sns.pairplot(inp1[['Reviews', 'Size', 'Price','Rating']])
plt.show()
```

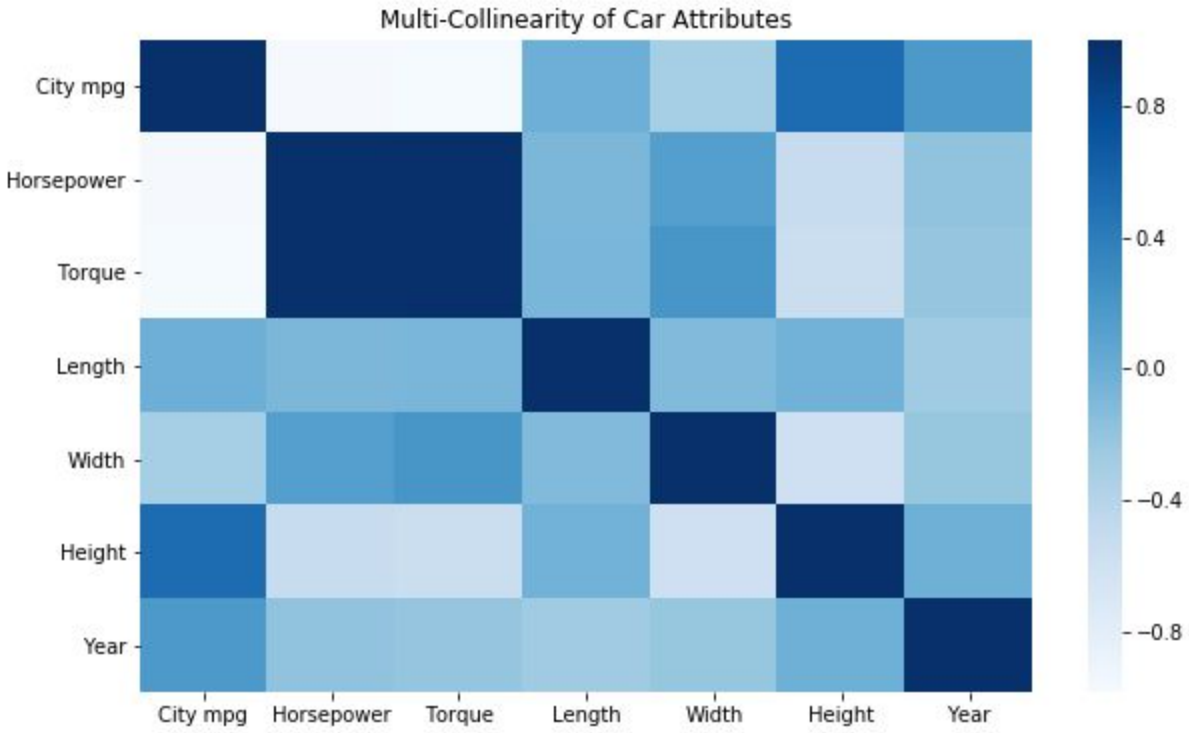


- When you have several numeric variables, making multiple scatter plots becomes rather tedious. Therefore, a pair plot visualisation is preferred where all the scatter plots are in a single view in the form of a matrix.
- For the non-diagonal views, it plots a **scatter plot** between two numeric variables.
- For the diagonal views, a **histogram is plotted.**

Heat Maps

Heat maps utilise the concept of using colours and colour intensities to visualise a range of values. In Python, you can create a heat map whenever you have a rectangular grid or table of numbers analysing any two features.

```
fig, ax = plt.subplots(figsize=(10,6))
sns.heatmap(data.corr(), center=0, cmap='Blues')
ax.set_title('Multi-Collinearity of Car Attributes')
```



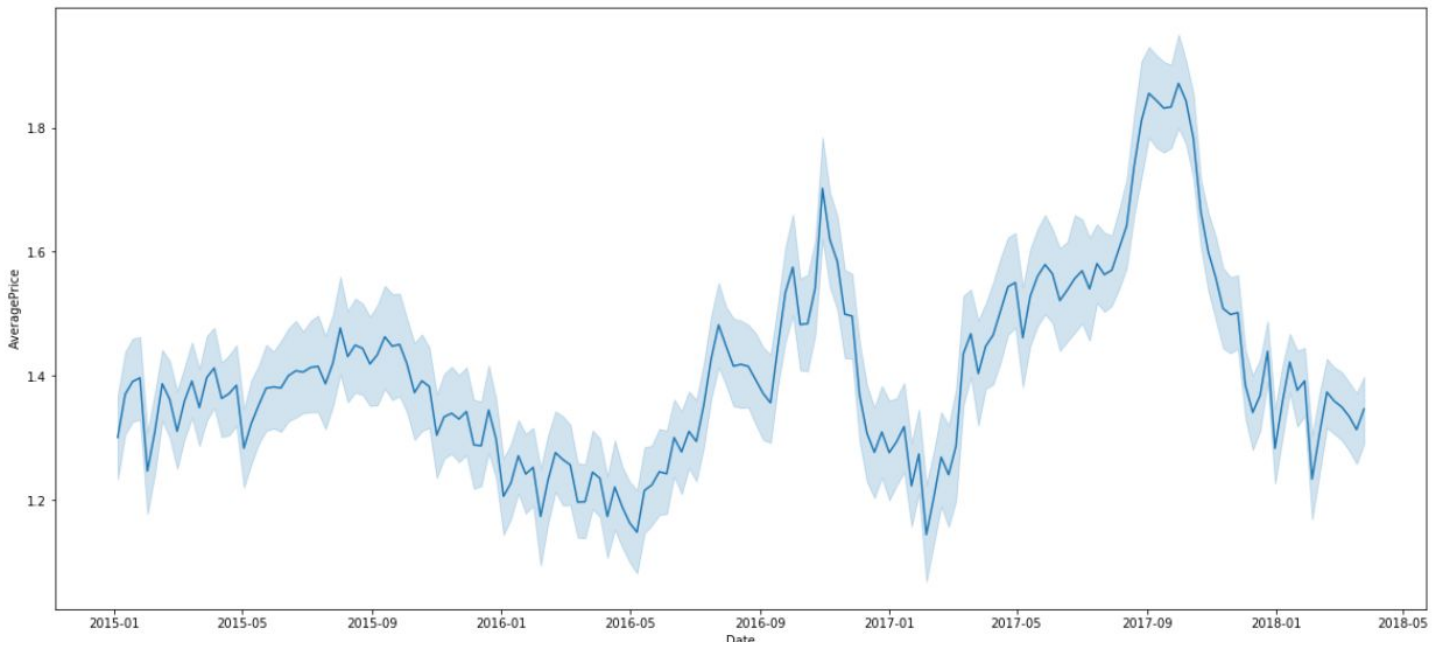
The heatmap shown previously represents the collinearity of the multiple variables in the dataset. The data.corr() function was used in the code to show the correlation between the values. This is where we want to set our independent or target variable. Looking at the blue heatmap, the focus should be on the dark and light areas. Dark blue represents a positive correlation, whereas white is a negative correlation.

Line Charts

A **line chart** or a line graph is a type of chart which displays information as a series of data points called markers connected by straight line segments. It is a basic type of chart commonly used in a number of domains.

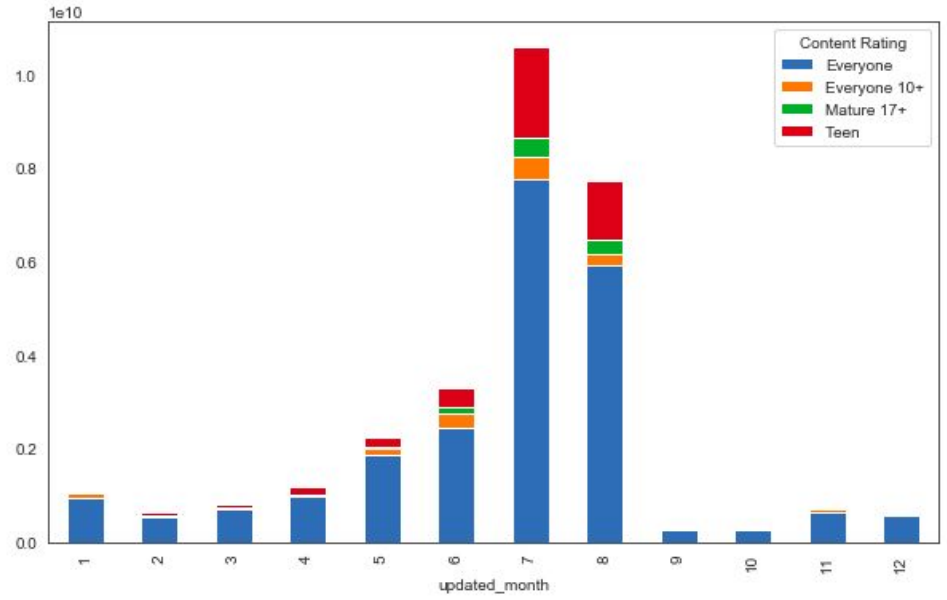
```
plt.figure(figsize=(20,9))
sns.lineplot(data=df, x='Date',y='AveragePrice')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2130a1d4be0>
```



Stacked Bar Graphs

```
##Plot the stacked bar chart.
monthly.plot(kind="bar", stacked="True", figsize=[10,6])
plt.show()
```



- A stacked bar chart breaks down each bar of the bar chart on the basis of a different category
- The main objective of a standard bar chart is to compare numeric values between levels of a categorical variable. One bar is plotted for each level of the categorical variable with each bar's length indicating a numeric value. A stacked bar chart not only achieves this objective, but also targets a second goal.

Plotly

The **plotly** Python library is an interactive, **open-source** plotting library that supports over 40 unique chart types covering a wide range of statistical, financial, geographic, scientific, and three-dimensional use cases.

Line Plot in Plotly:

```
#Import the plotly libraries
import plotly.express as px
```

```
#Prepare the plot
fig = px.line(res, x="updated_month",y="Rating",title="Montly average rating")
fig.show()
```